AFRL-RI-RS-TR-2013-180

HARDWARE ASSISTED STEALTHY DIVERSITY (CHECKMATE)

RAYTHEON BBN TECHNOLOGIES CORP

SEPTEMBER 2013

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2013-180 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /

SERGEY PANASYUK
Work Unit Manager

/ S /

MARK H. LINDERMAN
Technical Advisor, Computing
& Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
SEPTEMBER 2013	FINAL TECHNICAL REPORT	FEB 2012 – FEB 2013

4. TITLE AND SUBTITLE

HARDWARE ASSISTED STEALTHY DIVERSITY (CHECKMATE)

5a. CONTRACT NUMBER
FA8750-12-C-0098

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62788F

6. AUTHOR(S)

Joshua Edmison and Hina McCree

5d. PROJECT NUMBER
T2SD

5e. TASK NUMBER
HS

5f. WORK UNIT NUMBER
SD

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Raytheon BBN Technologies Corp
10 Moulton Street
Cambridge, MA 02138

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2013-180

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2013-2607
Date Cleared: 4 JUN 2013

13. SUPPLEMENTARY NOTES

14. ABSTRACT

CHECKMATE hardens homogenous computing environments from attacks through massive diversification of application execution. The CHECKMATE proof-of-concept prototype achieves diversification by leveraging underutilized silicon in today's computer systems such as unused memory, extra processor cores, and other underutilized processors such as graphical processing units (GPUs). Specifically, CHECKMATE introduces diversity in application execution by weaving the execution of many unique but functionally equivalent instruction streams representing an application. By introducing diversity at execution time, CHECKMATE greatly increases the effort required by an adversary to mount an attack against a system with little impact on performance. A successful attack against a CHECKMATE-enabled system would require successfully guessing the correct mixture of instruction streams and architectures before they are chosen at execution time. The combinatorial explosion of possible execution paths and architectural variation makes a successful navigation very improbable even with prior knowledge of the system components or the ability to guess at high speed. CHECKMATE is applicable to a wide-range of applications from embedded systems to commodity devices and has been shown to exhibit quantifiable security benefits.

15. SUBJECT TERMS

Diversity, diversification, attack, prevention, hardware, instruction set architecture, processor architecture, exponential, platform hardening

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**SERGEY PANASYUK**
U	U	U	UU	51	19b. TELEPONE NUMBER (*Include area code*) **N/A**

TABLE OF CONTENTS

LIST OF FIGURES iii
LIST OF TABLES iv
1 SUMMARY 1
1.1 Quantification of Protection Benefits 2
1.2 Potential For Operational Deployment 3
1.3 Recommendations & Future Work 3
2 INTRODUCTION 4
3 METHODS, ASSUMPTIONS, AND PROCEDURES 6
3.1 Achieving Diversity through Heterogeneous, Remote Application Execution 6
3.1.1 Application Segmentation 6
3.1.2 Segmentation Experimental Setup 8
3.1.3 CHECKMATE Emulation Testbed 9
3.1.4 Remote Execution Interface 10
3.1.5 Segmentation Experiment and Results 14
3.2 Achieving Diversity via Synthetic Architectures 14
3.2.1 Instruction Encoding for Diversity 14
3.2.2 Implementing Instruction Encoding 16
3.2.3 Synthetic Architecture Experimental Results 22
3.3 Reducing the Attack Surface via Architectural Shifting 25
3.3.1 Implementation of Architectural Shifting 27
3.3.2 Instruction Multiplexing with QEMU 29
3.3.3 Combined Results of Synthetic Architectures and Architectural Shifting 30
4 RESULTS AND DISCUSSION 31
4.1 Key Finding 1: Exponential Increase in Protection due to Diversity 31
4.2 Key Finding 2: Attack Detection and Alerting 31
4.3 Key Finding 3: Attack Types Prevented 31
4.3.1 User Experience and Performance 32
5 CONCLUSIONS 33
6 RECOMMENDATIONS 34

6.1 Transition CHECKMATE to TRL5 and Deploy CHECKMATE at Small Scale in an Operational Environment 34
6.1.1 Example CHECKMATE Enterprise Deployment Scenario 34
6.1.2 Sample Roll Out 34
6.1.3 Active Exploit Scenarios 35
6.1.4 Alerting and Protection System 37
6.1.5 Autonomous Protection 37
6.2 Extend CHECKMATE to Operate on Additional Platforms 37
6.3 Develop Advanced Techniques for Achieving Synthetic, At-Scale Diversity in Future Systems 38
6.4 Additional CHECKMATE Implementation Extensions and Enhancements 38
7 REFERENCES 40
APPENDIX 41
A.1 Initial Experimentation 41
A.1 Synthetic Attack Development 42
A.2 Initial Findings 43
A.2.1 Attack Indicator 43
A.2.2 Architecture Limitations 43
A.2.3 Application Limitations 43
LIST OF SYMBOLS, ABBREVIATIONS AND ACRONYMS 44

LIST OF FIGURES

Figure 1. CHECKMATE Adversary Effort 2
Figure 2. Typical CHECKMATE Prototype Configuration 6
Figure 3. Emulated Multi-Architecture System with Segmentation 7
Figure 4. Application Segmentation Process 7
Figure 5. Original OpenSSH Operation 8
Figure 6. Segmented OpenSSH Operation 9
Figure 7. Emulated multi-architecture system 10
Figure 8. Heterogeneous Remote Hardware System 11
Figure 9. Remote System Call Interface Block Diagram 11
Figure 10. Common Static and Dynamic System Calls 13
Figure 11. System Wide System Call Breakdown 13
Figure 12. Instruction Encoding Process 15
Figure 13. Instruction Decoding Process 16
Figure 14. Identifying Literal Pools with Symbol Table 17
Figure 15. Original QEMU Instruction Execution Process 18
Figure 16. Modified QEMU Instruction Execution Process 18
Figure 17. Original Hardware Instruction Execution Process 19
Figure 18. Modified Hardware Instruction Execution Process 19
Figure 19. Xilinx Zynq-7000 Block Diagram 20
Figure 20. Zynq Memory Access Pathways 21
Figure 21: Hardware-based CHECKMATE Prototype 22
Figure 22: Typical CHECKMATE Experimental Setup 22
Figure 23. Instruction Encoding Average Attempts to Breach 23
Figure 24. Probability of Preventing Attack with Watchdog 25
Figure 25. Probability of Stopping Attacker with Increasing Number of Architecture Shifts 27
Figure 26. Instruction Multiplexing Block Diagram 28
Figure 27. Probability of Stopping Attacker with Increasing Frequency of Multiplexer Shift 29
Figure 28: Realtime CHECKMATE Experimental Results 30
Figure 29. Interpreted Code Injection Attack on Multiple Architectures 32
Figure 30. CHECKMATE Enterprise Deployment 35
Figure 31. Active Exploit Scenarios 36
Figure 32: Simple Multi-Architecture System 41
Figure 33. Normal SSH Authentication 42
Figure 34. SSH Authentication Under Attack 42

LIST OF TABLES

Table 1. CHECKMATE Protection Summary 2
Table 2. CHECKMATE Protection Summary 32
Table 3. Active Exploit Scenarios 36
Table 4. Additional CHECKMATE Research 38

1 SUMMARY

CHECKMATE (Hardware-assisted Platform Diversification for Secure Polymorphic Computing) represents a fundamental advancement in computer architecture that neutralizes an entire class of widespread attacks against monoculture computing environments through massive diversification of application execution.

The goals of the CHECKMATE effort, both of which were achieved, are:

1. Establish the effectiveness of the CHECKMATE approach in preventing attacks and evaluate performance
2. Develop a prototype CHECKMATE platform

The CHECKMATE prototype creates a very high degree of diversification by leveraging a combination of underutilized silicon in today's computer systems such as unused memory, extra processor cores, available compute cycles, and other underutilized processors such as graphical processing units (GPUs). CHECKMATE diversifies application execution by seamlessly weaving the execution of many unique but functionally equivalent application instruction streams across multiple, heterogeneous processors. The combinatorial explosion of possible execution paths and architectural variation exponentially increases the effort required by an adversary to mount an attack with little cost to the system. A successful attack against a CHECKMATE-enabled system requires guessing the correct mixture of instruction streams and architectures before they are picked by the system at execution time. As such, attacking a CHECKMATE-enabled system is very difficult even with prior knowledge of the system components, operation, and algorithms.

Unlike many defensive solutions, CHECKMATE provides an asymmetric advantage to the defender. Figure 1 quantifies the exponential effort imposed upon adversaries as well as how existing monoculture systems fail to prevent attack due to a lack of diversity. Table 1 summarizes CHECKMATE's effectiveness against major attack classes. CHECKMATE neutralizes machine code injection attacks representing approximately 50% of the reported vulnerabilities in the Common Vulnerabilities and Exposures (CVE) database. For most applications, CHECKMATE protection produces negligible performance impact and the application degradation is imperceptible to the user.

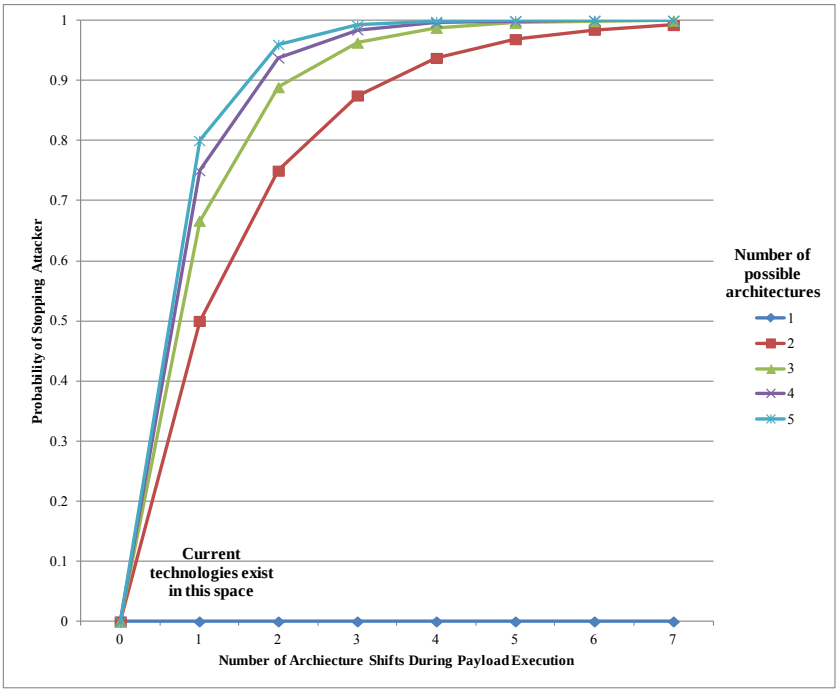

Figure 1. CHECKMATE Adversary Effort

Table 1. CHECKMATE Protection Summary

Attack Type	Protection
Machine Code Injection	CHECKMATE able to protect against
Return Oriented Programming	CHECKMATE offers limited protection
Interpreted Code Injection/SQL Injection	CHECKMATE cannot protect against

CHECKMATE also provides the ability to robustly detect attacks and attack attempts. Failed attacks against CHECKMATE are manifested as architecture mismatches or invalid instructions that are easily detectable. The low probability of successful attack combined with robust detection of attack attempts provides an opportunity to actively monitor, defend, or deploy countermeasures prior to a successful attack. The CHECKMATE capability is substantially different from monoculture systems where little or no opportunity exists to detect attacks through hardware operation.

1.1 Quantification of Protection Benefits

A major shortcoming of many computer software and hardware defensive mechanisms is the inability to provide a robust, quantitative assessment of protection. In contrast, the effectiveness of CHECKMATE's protection against attack is quantified through both empirical experiments and an analytical formulation. CHECKMATE's changes to computer architecture, such as the

addition of realtime, rapid switching between architectures called *architectural shifting*, provides a foundation for robust quantitative assessment of the CHECKMATE approach. A rigorous analytical model describing CHECKMATE's protection capabilities is described in Sections 3.2 and 3.3.

1.2 Potential For Operational Deployment

At the conclusion of the 12-month CHECKMATE effort both the software and hardware CHECKMATE prototypes are at Technology Readiness Level (TRL) 4. The CHECKMATE implementation of *synthetic diversity* through high-speed emulation provides a foundation for rapidly transitioning CHECKMATE to TRL 5 and beyond, as well as protecting operational commodity systems with the CHECKMATE approach. Synthetic diversity preserves the user experience and exhibits sufficient performance on commodity systems for applications that are not compute or memory bound. The current CHECKMATE implementation is tailored to Linux/Unix-based systems but the CHECKMATE concept fundamentally applies to all systems including Microsoft Windows, and Mac OS. In general, CHECKMATE is applicable to any system that has the ability to emulate another processor architecture or incorporate multiple physical processors. QEMU is an example of this type of capability. The CHECKMATE hardware implementation serves as a reference design for new systems.

1.3 Recommendations & Future Work

The following are recommendations for capitalizing on the findings and results of the CHECKMATE effort:

1. *Transition CHECKMATE to TRL5 and Deploy CHECKMATE at Small Scale in an Operational Environment*
   The initial research and demonstration of CHECKMATE has shown that the technology has strong potential to mitigate real threats to cyber infrastructure. If this capability can be transitioned to an operational environment, an asymmetric advantage could be given to defending forces. A key step in this would be the formalization of the implementation to determine its suitability to production systems. Deploying CHECKMATE in a limited test deployment provides an opportunity to develop CHECKMATE deployment methodologies as well as validate CHECKMATE protection under real-world attack situations.

2. *Extend CHECKMATE to Operate on Additional Platforms*
   Extension of CHECKMATE to other types of platforms such as single-purpose embedded systems promises to provide protection beyond general purpose platforms. Additionally deployment of CHECKMATE across an enterprise environment that

includes a mixture of general purpose platforms requires support of applications and operating systems such as Microsoft Windows, Apple OS X, iOS, and Android.

3. *Develop Advanced Techniques for Achieving Synthetic, At-Scale Diversity in Future Systems*
   Further research to develop additional techniques for achieving diversity as well as neutralizing additional attack classes such as attacks on interpreted code execution would extend the operation envelope of the CHECKMATE technology. As shown in CHECKMATE, multiple diversification techniques are needed to ensure robust protection. A suite of diversity techniques that leverage emerging technologies will ensure that diversification-based protection can be readily applied to future systems.

2 INTRODUCTION

Monoculture computing environments are susceptible to attack because of the commonality of the hardware and software across a large number of processing systems. In a monoculture environment a single vulnerability enables potential widespread harm and is attractive to attackers due to the large return for minimal effort.

For example, desktop computing is dominated by x86 processors, while mobile computing is dominated by ARM processors. Monocultures extend further than the processor architecture and include operating systems, applications and software stacks. When an adversary plans an attack on these systems, little reconnaissance is needed to craft a suitable attack payload that can assault a large number of devices. When a new code injection vulnerability is discovered, it is generally assumed that the corresponding exploit that is released will be written in x86 machine code.

Raytheon BBN Technologies has developed and demonstrated CHECKMATE, a suite of protection mechanisms that use existing available computing resources to reduce the attack surface available to exploit the vulnerabilities of a computing monoculture. CHECKMATE protects systems by diversifying application execution in a manner not possible in existing systems.

Diversity is an effective defense against existing code injection attacks which are, by their nature, dependent on the homogeneous nature of computing environments. The reduced attack surface also alters the adversaries' cost-benefit ratio requiring the development of attacks that can survive the shifting of architectures and which can operate in multiple architectures. This alters the attack environment in a way that places the burden of addressing a geometrically growing surface on the attacker rather than the defender. In enabling scale diversity, CHECKMATE addresses a systemic security issue common to single-processor Von Neumann computing systems and provides a framework for *accurately quantifying* the security benefits.

To quantify the effectiveness of CHECKMATE protections, an analytical model was developed and confirmed with empirical results. Quantification also provides an opportunity to compare the effectiveness of CHECKMATE against existing technologies.

Research and development during the CHECKMATE effort focused on three major areas outlined below. The results from each area combine to comprise the CHECKMATE protection suite:

- Achieving diversity through remote application execution on heterogeneous hardware
- Increasing diversity using synthetic architectures
- Reducing the attack surface via architectural shifting

One approach to creating more diverse systems by using underutilized hardware is to enable application execution across heterogeneous hardware. The initial set of hardware mixtures explored included ARM, PowerPC, and x86. Specifically application segmentation and remote execution of segments on different hardware architectures was investigated. Application segmentation allows a standalone application to be broken up into several parts. The remote execution interface allows each of these segments to be executed on remote processors communicating with each other as if they were running on the same processor. A seamless approach to heterogeneous application execution preserves the original functionality of the application.

Synthetic architecture diversity provides a mechanism for scaling in situations where additional physical architectures are not available or infeasible. Many unique processors are generated and emulated in a single-processor system to approximate a large-scale heterogeneous system. To execute applications in an architecturally diverse environment, the well-understood process of instruction encoding is applied to portions of existing applications. Instruction encoding allows a commodity processor to execute multiple architectures by transforming the native instruction of the physical processor. Combined with heterogeneous application execution, synthetic diversity proved to be a robust mechanism for achieving diversity on single architecture systems.

To further improve upon the benefits of diversification derived from heterogeneous execution and synthetic diversity, additional research was directed towards reducing the attack surface. One successful approach to attack surface reduction is called architectural shifting. Architectural shifting is the computer architecture analog to frequency hopping in radio systems. Instead of rapidly switching frequencies, architectural shifting seamlessly weaves many unique but functionally equivalent application instruction streams together. Many architectures combined with frequent shifts between architectures during execution, produces an exponential quantity of attack paths that nearly eliminate any chance of a successful attack.

A typical CHECKMATE enabled prototype system configuration is shown in Figure 2.

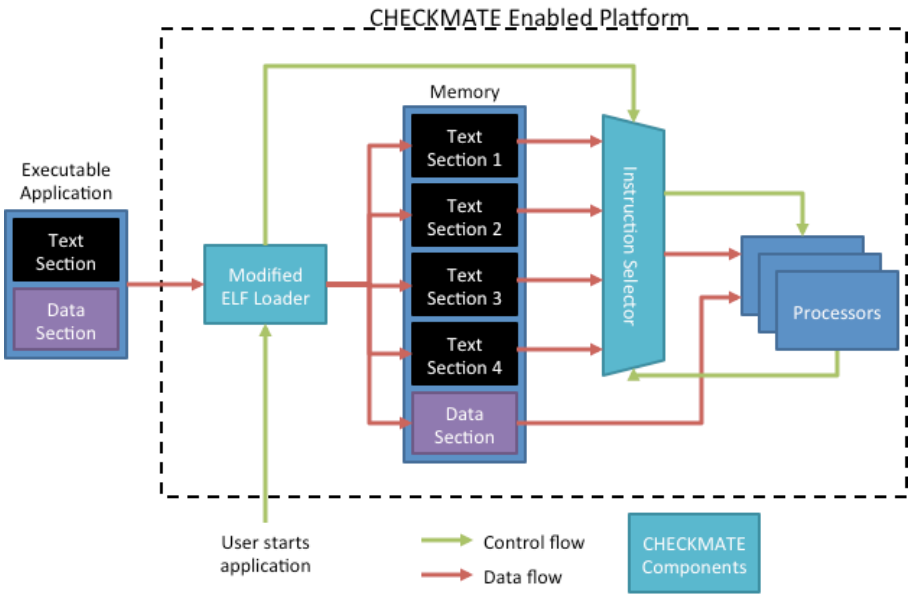

Figure 2. Typical CHECKMATE Prototype Configuration

The remainder of the document details design, development, implementation, and evaluation the CHECKMATE protections. Section 3 describes the methods, assumptions, and procedures employed throughout the CHECKMATE effort. Section 4 summarizes the results and findings of the CHECKMATE effort. Section 5 contains concluding remarks and Section 6 contains recommendations for future adaptations and extensions of the CHECKMATE protections.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Achieving Diversity through Heterogeneous, Remote Application Execution

One method to achieve diversity in a homogeneous system is to leverage underutilized hardware that is already present in the system. Examples of underutilized hardware include extra memory, extra compute cycles, unused cores, or other co-processors that are not used in the current system setup. To facilitate execution across these extra resources, particularly extra processors, applications must be split up to allow an application to run on multiple architectures at the same time. To achieve seamless execution and preserve original, standalone application functionality each segment must have a pathway for communication to other executing segments.

3.1.1 Application Segmentation

To increase the number of architectures that a single application can run on, a process of decomposing applications into segments was investigated. Application segmentation allows parts of a single application to run in multiple architectures at the same time, also allowing parts of an application to change architecture over the lifetime of a running program.

Figure 3 shows how application segmentation is setup on an emulated system. Each segment is compiled in the various architectures that are supported, and packaged as a standalone application. The combined operation of each segment forms the full functionality of the original application. As such, each segment is dependent on the other running segments. These application segments communicate with each using inter process communication (IPC) in the places where the original program shared data directly via a shared address space.

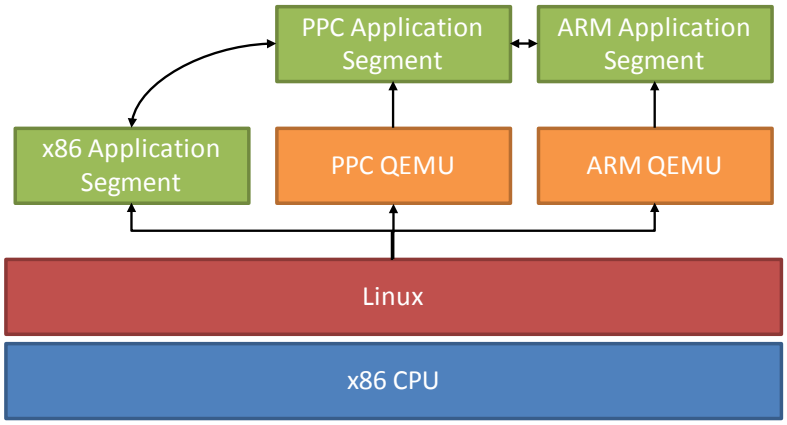

Figure 3. Emulated Multi-Architecture System with Segmentation

Several techniques for performing application segmentation were explored during the CHECKMATE effort. Automating segmentation along natural application boundaries such as basic blocks, function calls, or thread instantiation proved to be largely intractable. The difficulties in automating segmentation revolved around the need to understand from a security standpoint, which segments were of interest to an attacker, particularly sensitive, and/or vulnerable. Instead, manual approaches to segmentation were explored. The effort required to manually segment an application is dependent on three dominant factors. These are:

- An application that already contains independent processes that communicate can easily be converted, because each of the original processes can simply become a standalone application. Figure 4 outlines the details of the conversion process.

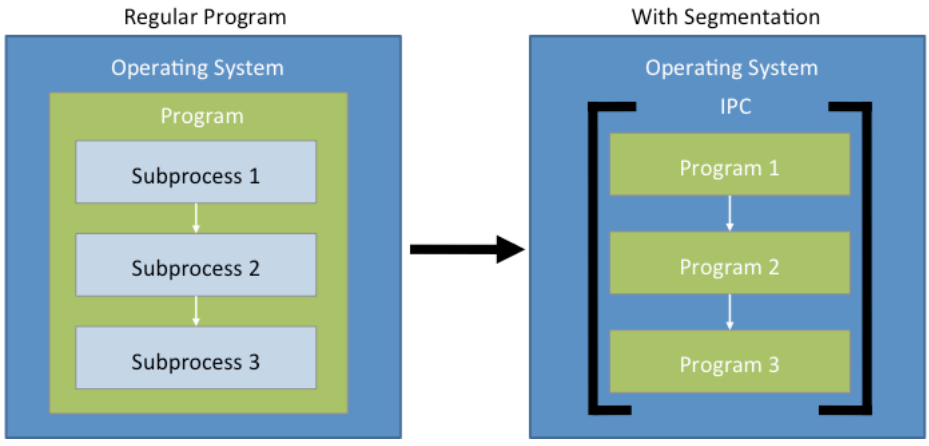

Figure 4. Application Segmentation Process

- Applications that have a single thread, and rely on shared access to a large amount of data are the hardest to segment, due to the connected nature of the application. The segmentation of applications like these relies heavily on IPC mechanisms such as shared memory to allow each segment to access the data it needs.
- The choice of where and how often to segment is also a consideration. Finer grained application segmentation reduces the likelihood of an attacker determining the target architecture for the attack since portions of the application will be running in different architectures at different times during execution.

Manual segmentation at the source code level, performed by a developer, programmer, or third party produced insight regarding the potential impact of diversified execution. Namely, that diversification can prevent attacks; however, the scale of the diversification is a key factor in preventing attacks.

3.1.2 Segmentation Experimental Setup

To evaluate the effectiveness of segmentation for attack prevention, an experiment was constructed that emulated typical attacks against a network-based application.

The test application, OpenSSH Server Daemon [1], spawns a new process each time a session is created. It is possible for multiple sessions to exist at once, but all remain inside the same application. Once the new session is started, each process is fairly standalone, and does not require interaction with the main process, or the other sessions. Figure 5 illustrates the per-session thread spawning process of OpenSSH.

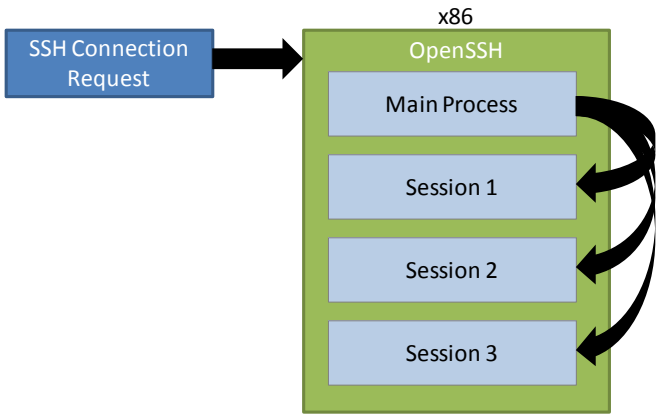

Figure 5. Original OpenSSH Operation

To demonstrate the technique of application segmentation, OpenSSH was segmented at a session boundary. Each time a new user logs in; a new OpenSSH segment is started as a new application on a random architecture. Figure 6 shows how multiple sessions may be running at the same time, each in their own architecture. If an attacker were to craft an attack payload for one particular

segment, it would not be effective on the other architectures, and the attacker would not be able to predict when a given architecture will run.

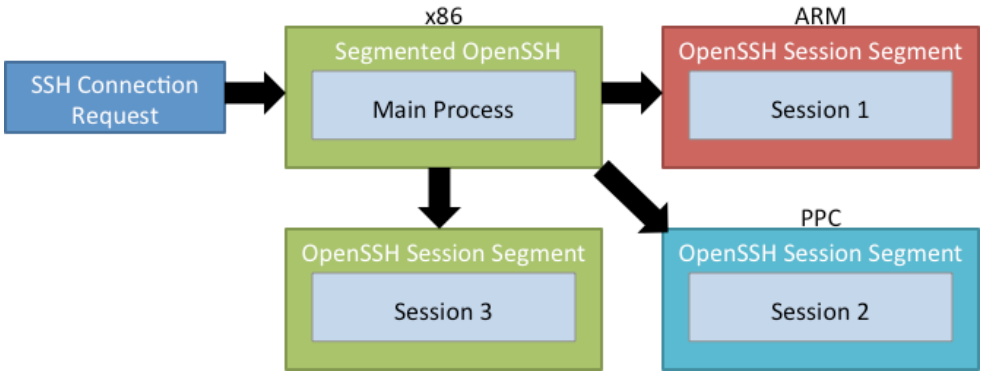

Figure 6. Segmented OpenSSH Operation

While segmentation was demonstrated using OpenSSH, the concept of segmentation is applicable to nearly any application. Additional experiments were performed with Mongoose [2], a lightweight web server. Mongoose was modified to start execution with a new architecture for each new HTTP request. A similar approach is relevant to most server applications that handle multiple disjoint requests; each request can be handled by a different architecture. To demonstrate the applicability of segmentation to a non-networked application, gzip [3] a compression utility, was also segmented such that each time a file is compressed the compression calculations are performed on a different architecture. No graphical applications were included in the experiments; however typical thread-base handling of graphics provides a natural boundary for segmentation and diversification. One example would be to render each window on a unique architecture.

3.1.3 CHECKMATE Emulation Testbed

One challenge when exploring heterogeneous architectures is the difficulty of co-existing and interfacing multiple instruction set architectures. Several physical testbed alternatives were evaluated for combining x86, PowerPC, and ARM processors into a single physical system. Given that nearly all of the COTS platforms incorporating these processors were intended for standalone use, integrating them into a single system where the individual processors could interact proved both costly and time-consuming. Another approach, high-speed emulation provided the ability to evaluate the effectiveness of CHECKMATE across, not only many architectures, but also many instances of each architecture. High-speed emulation also provides a mechanism for retrofitting architecture diversity onto systems that use single processors by leveraging the extra compute cycles and system memory. The popular QEMU emulation

platform was chosen as the CHECKMATE emulation platform for its speed, robustness, and widespread use.

The QEMU emulator is able to solve both the problem of system availability and system communication. The QEMU emulator is an open source processor emulator that is able to emulate multiple architectures (including ARM and PPC) and is capable of running in multiple modes of operation. First, it is able to create a traditional virtual machine, where all system resources are emulated (disk, processor, RAM). Secondly, it is able to emulate a processor on a per application basis, sharing the resources of the host system, and performing architecture specific translation when necessary. The system resource sharing mode of QEMU was used to create the first testbed, because it is able to satisfy both the application communication and system availability problem. An additional benefit of the emulation-based test bed is the ability to quickly scale the number of architectures in the system.

The initial emulated test bed is show in Figure 7, which consists of an x86 system containing an ARM QEMU and PPC QEMU instance capable of running applications of their respective architecture. With control centralized in the x86, a simple application launcher was created that would randomly choose which architecture to run x86, ARM or PPC.

Figure 7. Emulated multi-architecture system

The first application tested was OpenSSH, an open source version of the SSH connectivity tools including both a client and a server that provide user terminal access to the server upon client authentication. OpenSSH represents a non-trivial, network-connected application that is commonly attacked.

3.1.4 Remote Execution Interface

Initial tests using the emulated testbed indicated that sharing of local resources from the user interface systems is key to enabling CHECKMATE in a transparent manner. QEMU provides a resource sharing mechanism when all segments are running on a single processor; however a similar functionality is required when using remote hardware [4]. In this case, remote hardware refers to the processors that *do not share a memory subsystem with the controller.*

A mechanism to support access to local resources from applications running on remote hardware is required. Figure 8 illustrates operation of the remote interface. To facilitate remote operation, an interface needs to be identified that allows interface calls to be intercepted and sent to the interface system.

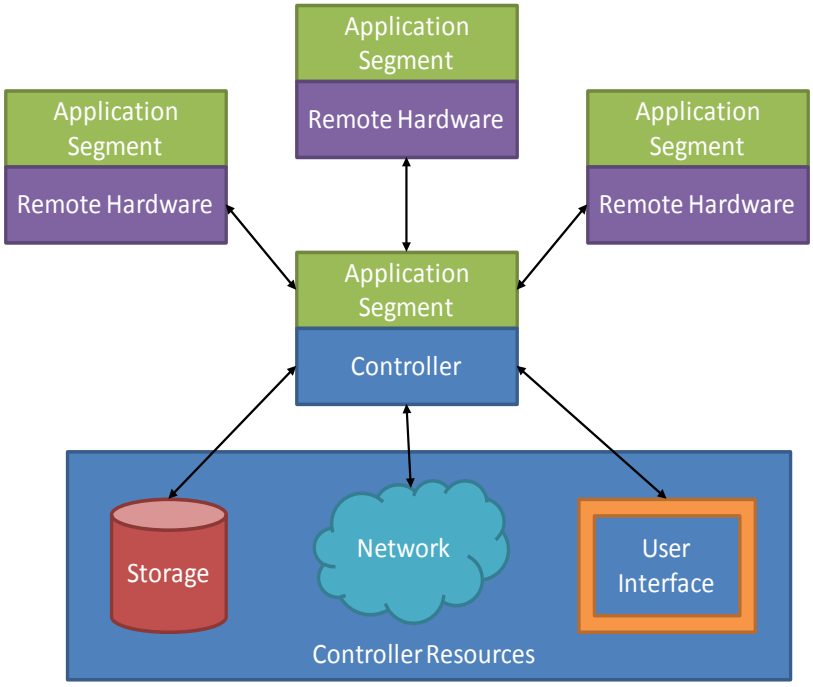

Figure 8. Heterogeneous Remote Hardware System

In the case of Linux, most interaction with the outside world goes through the kernel. Since the applications that CHECKMATE will be protecting reside in user space, the system call interface that separates user space and kernel space is an ideal candidate for an interception point. The implemented remote system call interface is shown in Figure 9.

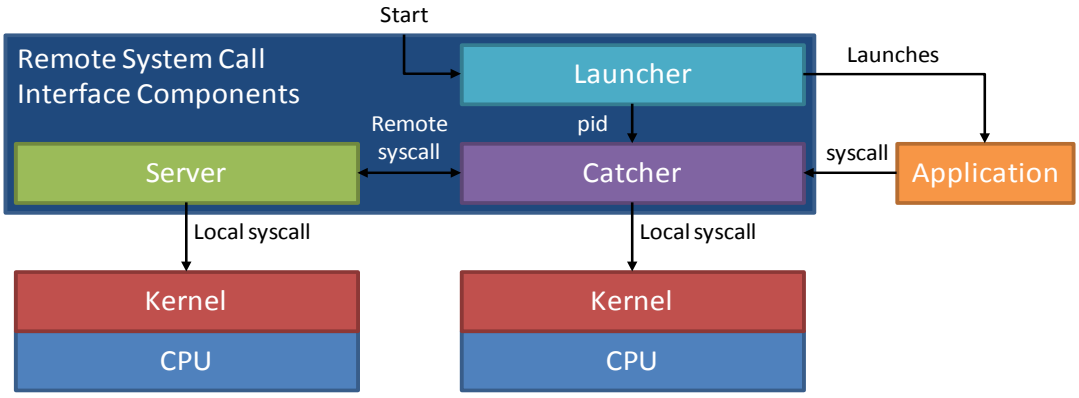

Figure 9. Remote System Call Interface Block Diagram

The remote system call interface contains three major components:

- **Server**: runs on the interface machine, the point where the user or an external network is connected to the system (the "Controller" from Figure 8). The server connects to remote machines and accepts system calls as applications make them on remote hardware. These system calls are then executed locally and the results are returned back to the remote hardware.
- **Catcher**: a kernel module that runs on the remote machines, that is responsible for catching system calls, and then communicating state information to the server (if required). Once the system calls have been completed, the catcher collects the results and returns them to the calling application.
- **Launcher**: launches the application that needs to use the remote system call interface, informing the catcher which process ID to catch system calls for.

3.1.4.1 Identifying Interface Boundaries

All system calls cannot be blindly sent over the interface. The local kernel is still responsible for the memory management of the running application, so the catcher needs to filter out the system calls that need to be handled locally. However, system calls can be roughly sorted to different classes (based on their arguments), so an interface generator was created that is capable of automatically generating interface code for the most commonly used system calls.

There are over 300 system calls in the 3.x version of the Linux kernel. Not all system calls are actively used in most applications. Analysis of system call usage across common applications determined which system calls are necessary to support cross-architecture execution.

Figure 10 shows the common system calls for OpenSSH (calls made or referenced more than 5 times) when both static and dynamic analysis is performed. To support OpenSSH with the remote system call interface the system calls in Figure 10 are required. Analyzing both the static and dynamic call usage is important as either type of analysis alone incorrectly quantities overall call usage throughout application execution lifetime. For example, only looking at the static analysis would have given the impression that `futex` was more common than `open`, when in fact `open` is called much more often.

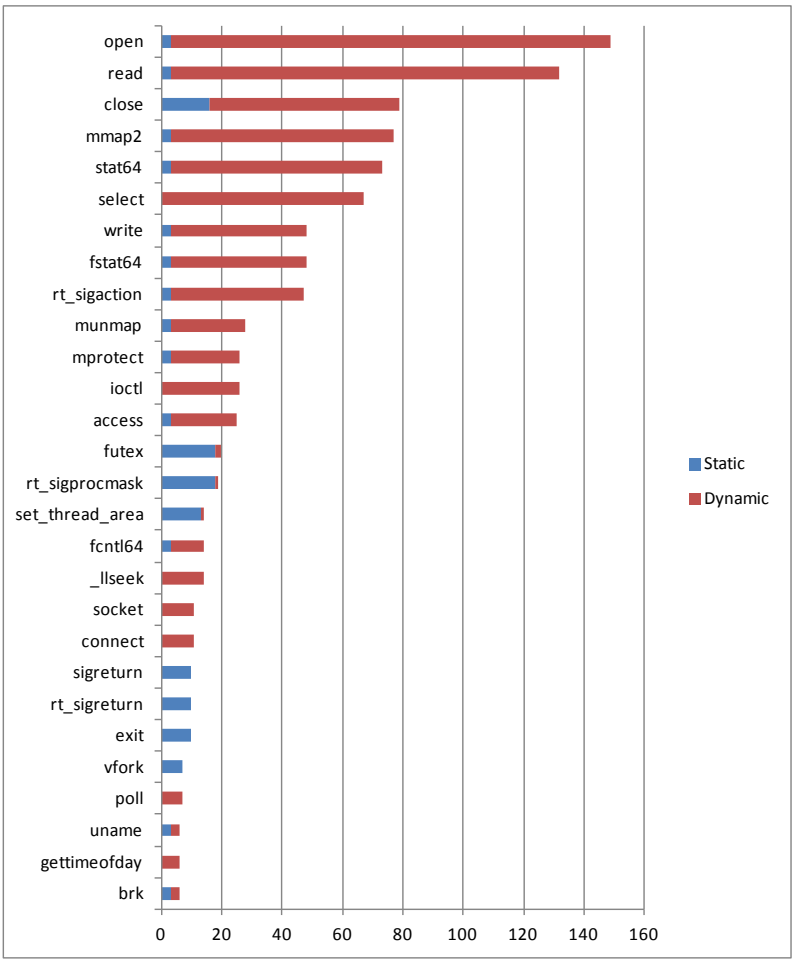

Figure 10. Common Static and Dynamic System Calls

The combined static and dynamic analysis was performed on a set of common Linux binaries in order to understand the usage of system calls across common applications. The analysis shows a similar breakdown as for OpenSSH. Of the 66 total calls, 55 were called 100 times or less. The breakdown of the remaining is shown in Figure 11.

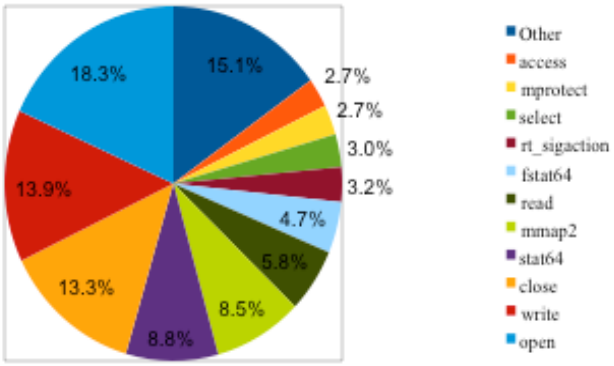

Figure 11. System Wide System Call Breakdown

While the analysis was performed for the Linux system call interface, the analysis process is sufficiently general to apply to a range of common hardware abstraction layers (POSIX, etc.). The idea is to identify all possible interface points and the likelihood of each in order to determine which information needs to be transferred from the remote hardware to the server.

3.1.5 Segmentation Experiment and Results

To evaluate the effectiveness of segmentation-based diversity, attacks were performed against the segmented and non-segmented versions of OpenSSH. To simplify the experimental process, vulnerabilities were inserted into OpenSSH as discussed in Appendix A.1. Multiple versions of the attacks against the inserted vulnerabilities were created.

Attacks against a segmented application succeeded when the instruction-set architecture used in the attack matched the selected architecture for the OpenSSH session. Given the small count of three architectures, the likelihood of a successful attack was approximately 33%. Failed attacks are manifested as illegal instructions that are easily detectable. Segmentation demonstrates the potential for diversity to prevent attack however segmentation with a small number of architectures does not alone provide sufficient protection. To address the lack of architecture diversity, additional methods for expanding diversity beyond the base architecture(s) found in a system were explored.

3.2 Achieving Diversity via Synthetic Architectures

The results of the segmentation experiment indicate that diversity is effective for preventing attack but that substantial architecture diversity is required for effective attack prevention. In systems with a limited number of architectures, methods for artificially expanding architectural diversity are necessary. One approach to establishing many unique architectures on a system is a concept called *synthetic architectures*. By utilizing extra processing power and extra memory, numerous synthetic architectures can be created and used to diversify execution. Synthetic architectures can be created from scratch in either actual hardware or through emulation. These architectures consist of new opcodes and instruction formats. The remote execution infrastructure discussed previously provides a means for spreading execution across these resources. Another, potentially more efficient approach, to creating synthetic architectures is a technique called instruction encoding.

3.2.1 Instruction Encoding for Diversity

To supplement the physical and emulated architectures, the concept of instruction encoding was developed. Unlike instruction set randomization [5] that attempts to encode instructions *a priori* with one or more static keys, instruction encoding encodes instructions on the fly without prior knowledge of a key to enable a commodity processor to execute many unique instruction sets. As such it is possible to synthetically increase the number of architectures in a system for

purposes of diversification. Multiple encodings in a single system where one or more base architectures coexist with multiple encoded architectures serves as the basis for synthetic architectures. Synthetic architectures allow provide a mechanism for bypassing the limit of physical architectures at minimal computational cost.

There are two parts to the instruction encoding process. First, the encoder takes a set of instructions for a particular architecture and applies a reversible one-to-one function on that instruction. The encoding function needs to be one-to-one because it needs to be possible to get back to the original instruction using the correct decoding function. The encoding value provides a means for differentiation during both the encoding and decoding process. The instruction encoding process is illustrated in Figure 12.

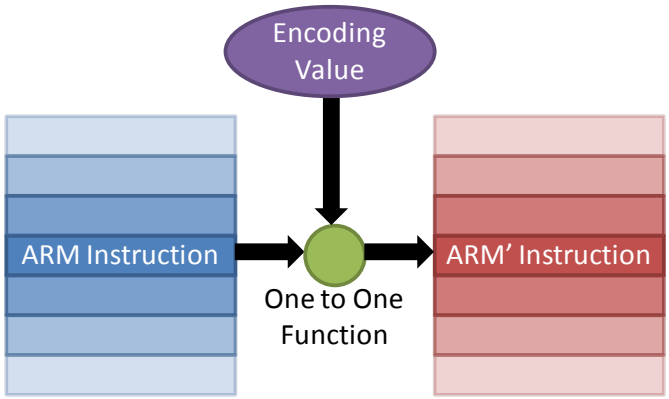

Figure 12. Instruction Encoding Process

The encoding process is similar to encryption, but it is important to differentiate it, because the goals are quite different. In the case of encryption the goal is confidentiality, meaning that the contents of the instruction and its purpose are being hidden. Also, instruction encoding bares some semblance to instruction set randomization but with a key difference that the encoding occurs on-the-fly at runtime and not with prior knowledge of a key value. The lack of one or more static key values is a key enabler for diversity and avoids the need for processor support or a key management scheme. With instruction encoding the goal is diversity, and there is no need to keep the encoding value a secret (unlike an encryption key). In addition, the encoded and un-encoded applications are likely to live in the same location. This is ill-advised if the goal was confidentiality.

The instruction is decoded as it is fetched from memory and transferred to the processor. CHECKMATE is concerned with defending code injection attacks that occur as the program resides in memory. By decoding the instruction as late as possible, most code injection attacks can be prevented. To simplify the experiment setup, it is assumed that the attacker is not aware of CHECKMATE protections, and the injected attack instructions are not encoded. Note however that CHECKMATE protections are not dependent upon limited attacker knowledge of CHECKMATE's techniques. Figure 13 shows the decoding process, where the decoding value

is the same value used from the encode process, and the decode function is the reverse of the encode function. Decoding only occurs if an instruction fetch is being performed; all other memory accesses remain unmodified.

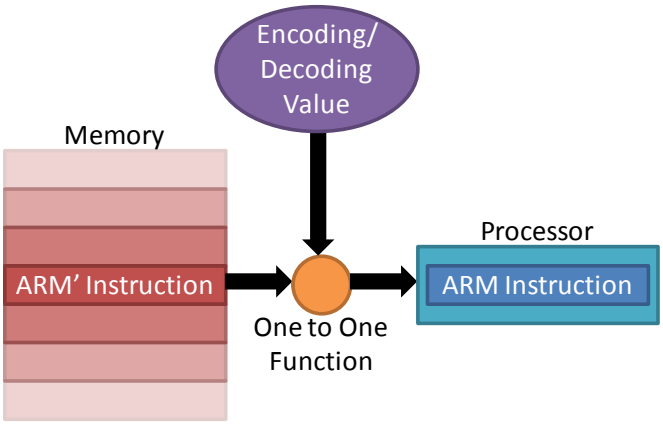

Figure 13. Instruction Decoding Process

3.2.2 Implementing Instruction Encoding

Two major challenges were discovered while implementing instruction encoding. First, on the encoding side, all the instructions of an application need to identified and encoded, but the data and other control information avoided. ARM was chosen as the base architecture due to its fixed width instructions and wide availability of flexible development boards.

The second challenge of instruction encoding, is inserting the decode block in between the instructions residing in memory and the processor. Usually the memory and processor (with cache) would be directly connected with no opportunity for interception, but the choice of emulation platforms allows an opportunity to get access to the processor-memory boundary. Implementing instruction encoding in hardware requires a more creative approach, discussed in section 3.2.2.4.

3.2.2.1 ARM Executable Encoder

The encoder takes as input, a standard ELF (Executable and Linkable Format) executable and an encoding value. Using the ELF headers, it finds the ELF for its text section (executable instructions), encodes each instruction with the encoding value, and outputs the encoded executable. An exclusive-or (XOR) is the current encoding mechanism, as it is simple to implement, and easily reversible by a decoder.

While ARM uses a relatively simple RISC instruction set, and commodity hardware is readily available for it, ARM architectures have an interesting feature, not shared by x86 and PowerPC, which required one additional step in the encoding process.

ARM makes use of "literal pools" within text sections of executables. Literal pools are generally small areas of data within a larger stream of instructions. Compilers for true RISC processors such as ARM, insert literal pools such that a single instruction can reference the data (using an

immediate offset for example), instead of requiring two instructions. Literal pools present a problem for encoding since only instructions and not data will be decoded. If left in an encoded state, any literal pools will be improperly decoded and result in executable malfunctions.

To combat the literal pool problem, several identification methods were investigated. First, a modification to the compiler tool-suite "binutils" was considered. Specifically, update the "gas" (GNU Assembler) to emit modified instructions. The assembler approach is applicable to approximately 95% of instructions; however some instructions do not exist until link time or are not fully resolved. Branches outside of the local object file are one example of instructions that are missed by the assembler approach to the literal pool problem. Modification of the linker (ld) is another approach to obtaining the necessary information; however linker modification was complex and forced modification of standard application development tools.

Next, the GNU objdump tool was examined as a potential source of information that could mitigate the literal pool problem. Using objdump, it is possible to generate disassembled output of the binary. From the disassembly it is possible to distinguish executable code from data, including data embedded in the executable text segments. The objdump tool provided reliable results for the majority of instructions; however, not all instructions are properly identified and 100% identification is required for a program to work correctly.

The relocation header of the ELF executable as also investigated. Using gcc --emit-relocs, the compiler adds a special section in the final executable known as relocations. By scanning relocation headers, it is possible to identify some of the data in the text section. In this technique, many elements are missed, which also makes this approach infeasible.

Finally, using the symbol table section of the ELF executable was attempted. This technique was ultimately chosen since most assembler/compilers emits a special symbol ($d) every time it inserts a literal pool (data in the text section) into the image. Additionally, another symbol ($a) is added once the instruction stream resumes. A representation of literal pools and their relationship to their symbol table is shown in Figure 14. By using symbol tables, we can identify all of the data in the instruction stream. A minor challenge is that the size of the literal pool is not added as part of the symbol, so it is required to scan forward in the executable until the next instruction symbol ($a).

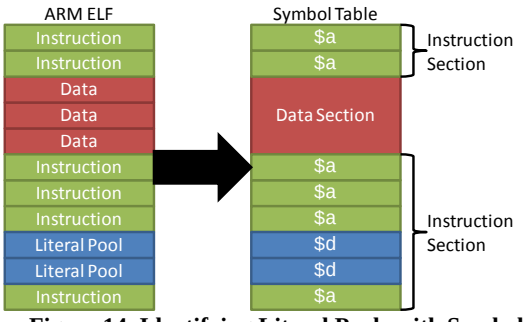

Figure 14. Identifying Literal Pools with Symbol Table

3.2.2.2 Instruction Encoding/Decoding with Emulation

The QEMU emulator used during initial experimentation provides full access to all the emulated hardware components through modification of the QEMU source code. The original QEMU instruction execution process flow is show in Figure 15.

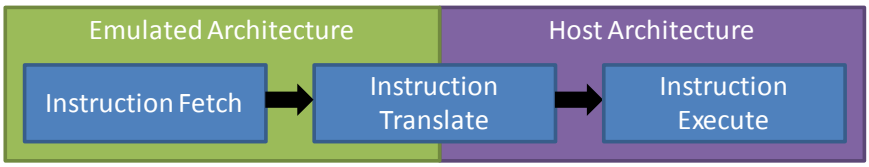

Figure 15. Original QEMU Instruction Execution Process

The emulated architecture instruction is fetched from memory, translated into an instruction (or set of instructions) for the host architecture, and then executed. By injecting a decode step into the instruction pipeline, instruction decoding can be performed in QEMU.

Figure 16 shows the modified instruction fetch process. The instruction decode is added after the fetch, and a new instruction state is added. Also demonstrated in Figure 16 is that the time that instructions exist in the "Emulated Architecture" state is kept to a minimum, as this period provides an opportunity to perform a code injection attack.

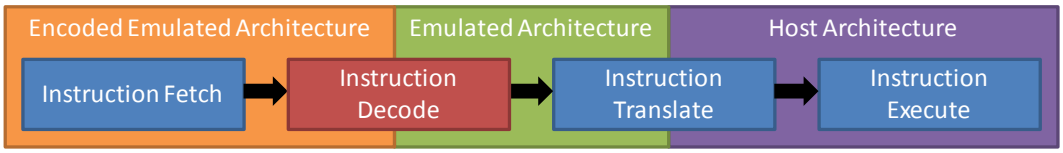

Figure 16. Modified QEMU Instruction Execution Process

3.2.2.3 Emulated Instruction Encoding Experimental Setup

Instruction encoding was added to the emulation testbed outlined in Section 3.1.3 allowing for an increased number of emulated architectures in a single system. The only limit to the number of architectures is the memory required to store the multiple copies of the encoded architectures, and the computation time it takes to generate the encoded applications. A maximum of 50 emulated synthetic architectures were used in the experiment.

Synthetic architectures are compatible with any host architecture supported by QEMU. During the experimentation the synthetic architectures were executed on both x86 host and ARM hosts.

3.2.2.4 Hardware Instruction Encoding

The emulator solution is flexible and easy to implement on multiple platforms, but has the drawback of running inside an emulator that significantly reduces application speed, as compared to native execution. Hardware encoding is able to achieve these native speeds, but requires additional hardware to do so. The normal instruction execution process when excluding cache interaction is shown in Figure 17.

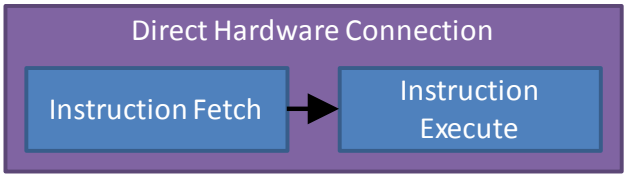

Figure 17. Original Hardware Instruction Execution Process

To implement instruction encoding, a method of intercepting the processor accesses to memory is required. While the processor's connection to memory is direct, it is also a bus, meaning that the processor is able to access other devices on the same bus. Bus-based memory access provides a means to implement an in-line decoder by adding a new device to the memory bus and rerouting processor memory accesses through the new device. In the CHECKMATE implementation the rerouting of memory access through the added memory decoder is achieved by remapping requests to a new address range.

The process of fetching an instruction through the decoder is illustrated in Figure 18. It should be noted that without extra processing, all accesses through the decoder will be decoded. Since data is not encoded, the decoder will need to be aware if a data or instruction access is occurring. To route processor accesses to the correct address, the MMU can be modified to access memory through the correct range. MMU redirection is implemented as a mask, so for example if the real memory base was `0x00000000` – `0x40000000`, the same memory could be accessed through the decoder at address range `0x80000000` – `0xC0000000`, meaning there is a fixed offset of `0x80000000`. Knowledge of the precise address scheme would not benefit the attacker because from the point of view of user space (the attackers perspective), the virtual address that is being accessed does not change, so there is no security benefit to a more complex addressing scheme.

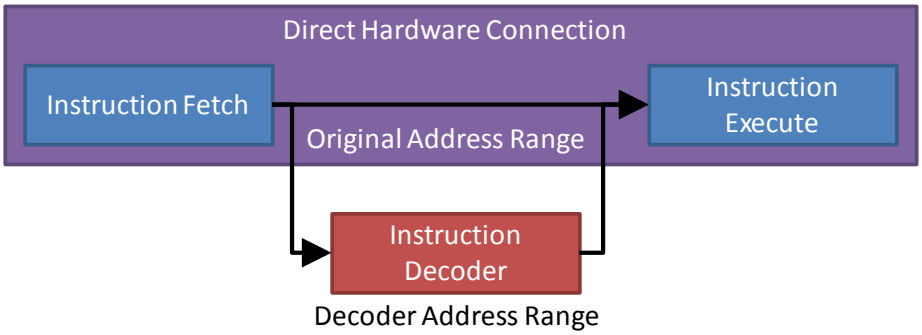

Figure 18. Modified Hardware Instruction Execution Process

3.2.2.5 Hardware Instruction Encoding Experimental Setup

Required for hardware decoding is a device that externally exposes the memory bus, as well as another processing element (processor or FPGA) that can act as the decoder. The Xilinx Zynq-7000 embedded on a Digilent ZedBoard was used for our experimental setup.

The Zynq provides dual core Cortex-A9 (same cores that are in the Tegra 3 and OMAP4400 series SoCs). Memory accesses are performed of the AMBA busses, but unique to the Zynq, the AMBA switches connect both to the memory controller and the programmable logic. AMBA access allows the processor to access the same memory region directly, or through the decoder, depending on the needs of the application that is currently running. Figure 19 shows the high level block diagram of the Zynq processing system. Of particular interest are the connection points between the programmable logic and the processing core.

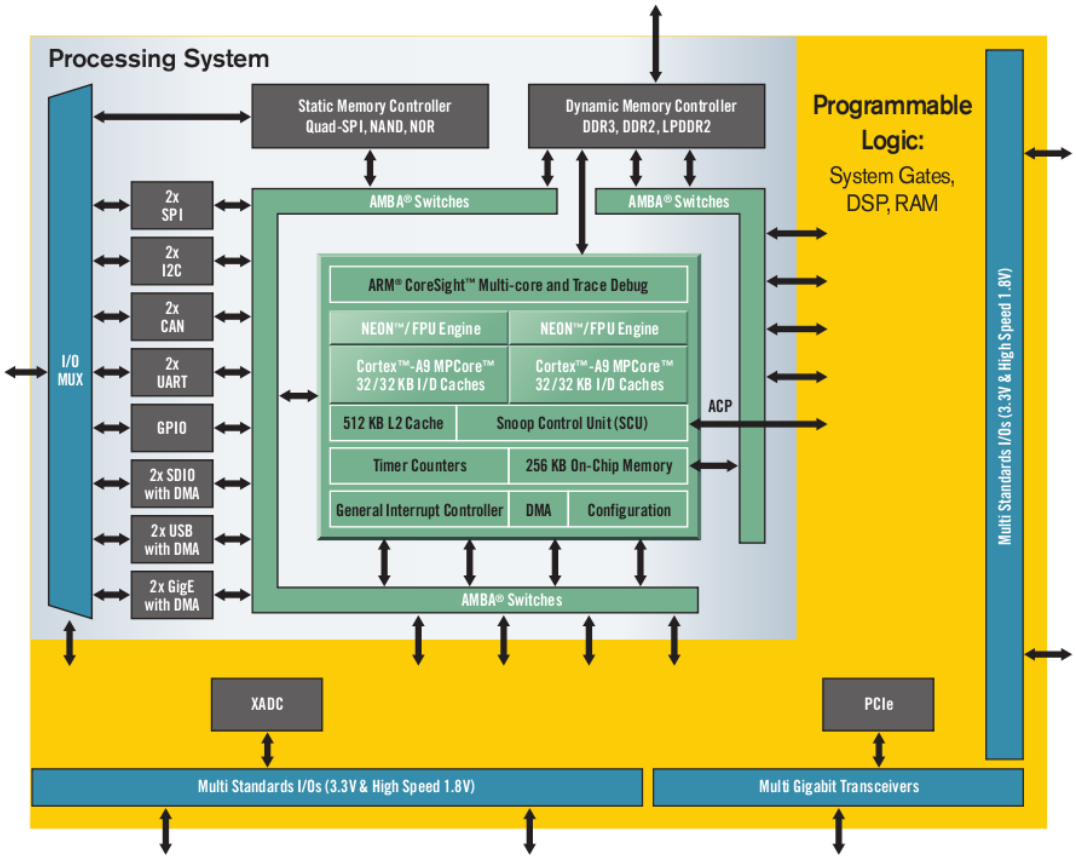

Figure 19. Xilinx Zynq-7000 Block Diagram

Figure 20 shows the different pathways that the memory accesses follow. When the processor accesses memory directly (via address range `0x00000000` – `0x40000000`) the access is routed through port M0, following the red path to the memory controller. The decoder is designed with a configurable address range, and when memory is accessed via the specified address range (`0x80000000` – `0xC0000000`), access is routed through port M1 following the green path. The resulting signal is routed through the master interconnect, and then up to the programmable logic, where the decoder resides. Once the memory access is received, the address is modified to its memory counterpart (fixed offset of `0x80000000` is subtracted). The request is then sent down through the AXI HP to DDR interconnect and finally to the memory controller. The data accessed travels back to the decoder (in its encoded state, if it is an

instruction). Once the data is at the decoder, it determines if the information being is accessed is an instruction. The AXI interconnect specification includes the AR_PROT signal which includes a bit that is set if the access is an instruction. Based on the AR_PROT signal, data access is ignored by the decoder and instructions are decoded for execution on the controller.

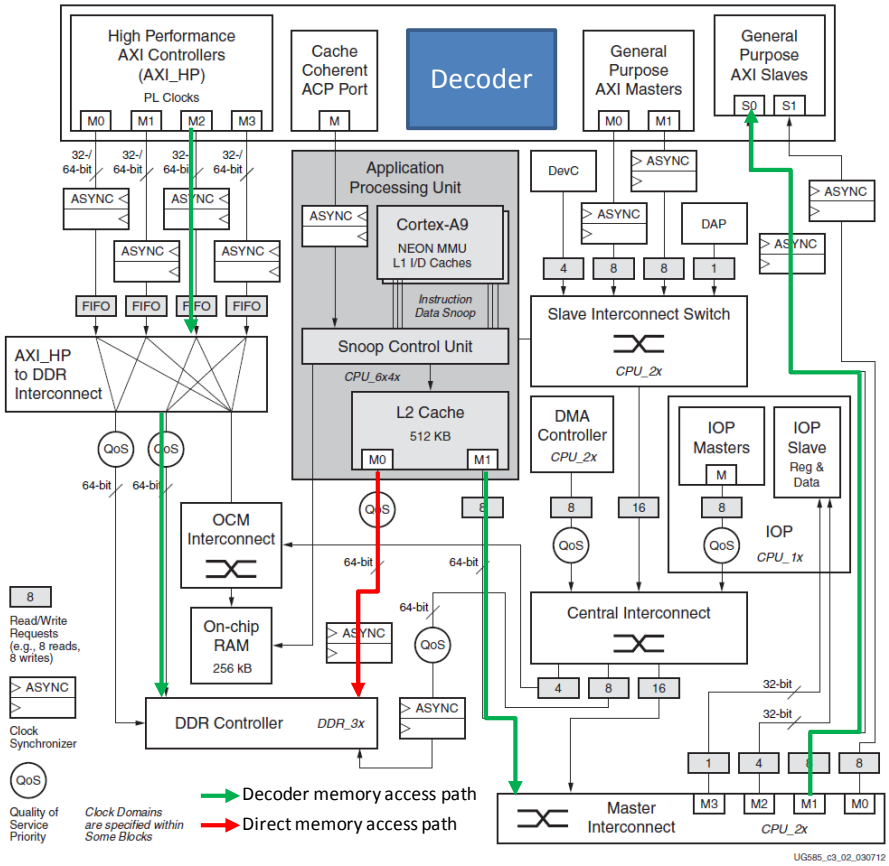

Figure 20. Zynq Memory Access Pathways

The hardware-based CHECKMATE testbed and typical experimental setup are shown in Figure 21 and Figure 22.

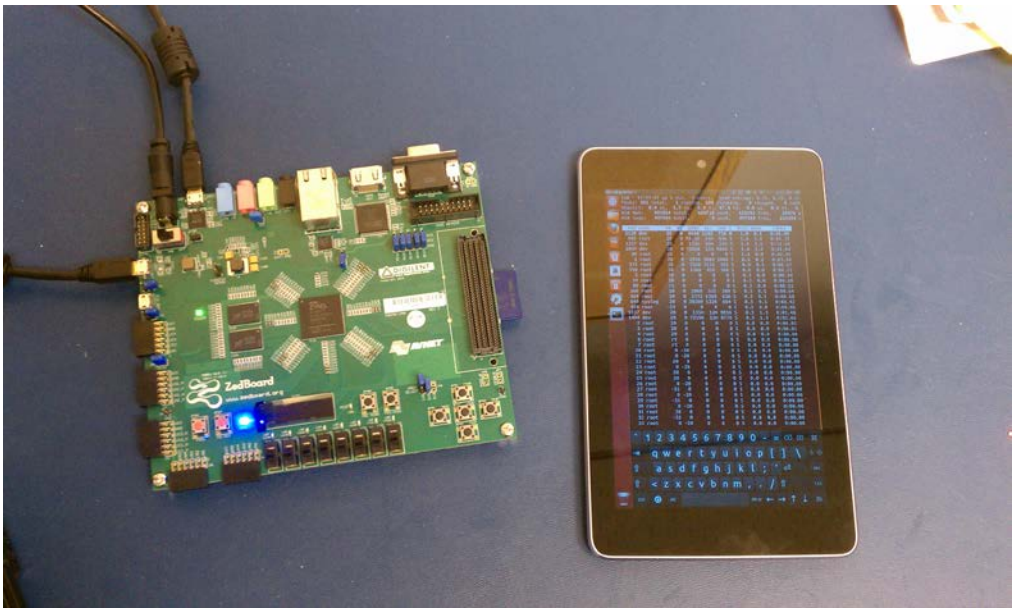

Figure 21: Hardware-based CHECKMATE Prototype

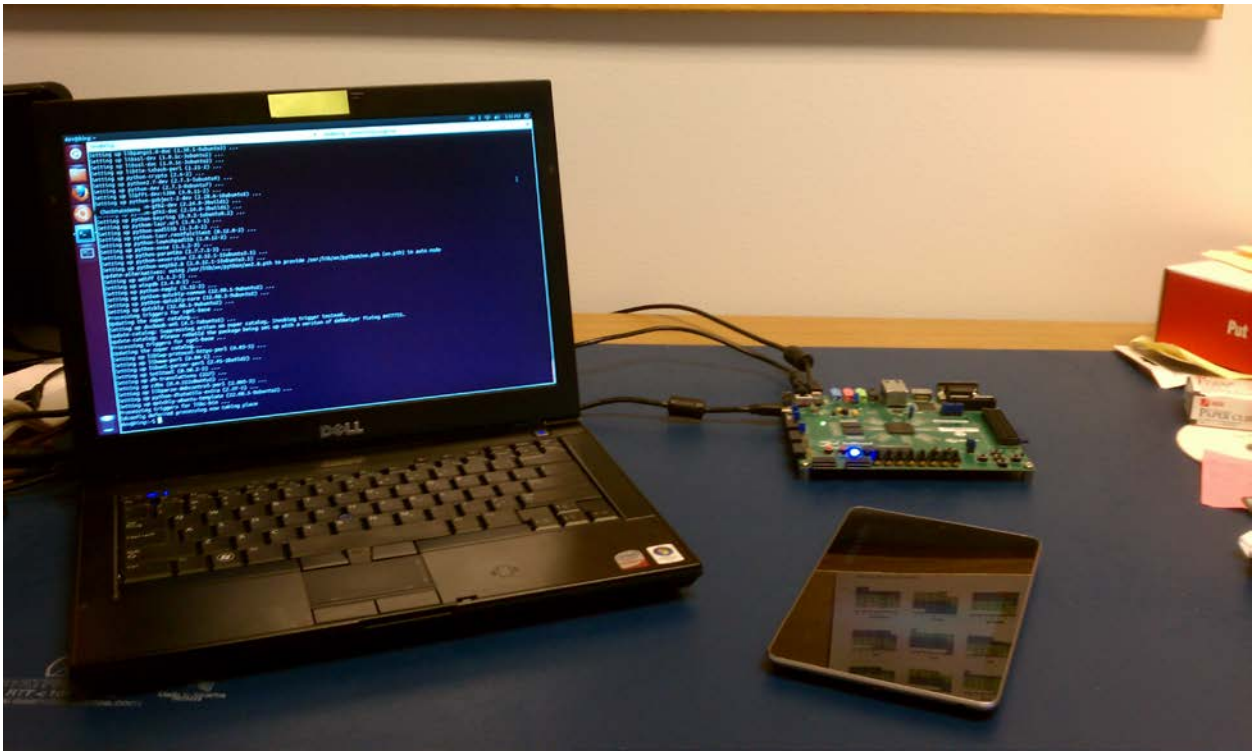

Figure 22: Typical CHECKMATE Experimental Setup

3.2.3 Synthetic Architecture Experimental Results

The OpenSSH test attacks used in the segmentation experiments were also used to evaluate the effectiveness of diversity through synthetic architectures. With the additional architectures

provided by instruction encoding, the OpenSSH test exploit from section A.1 was re-run. Tests were run with a range of virtual architectures, from 2 to 47 to demonstrate how the security benefits of additional architectures scale with added architectures. For each set of architectures, the exploit was run until a successful breach 100 times, and the average number attempts to breach were recorded. The result is a relative measure of security for the system. Experimental results for the all of the runs are shown in Figure 23.

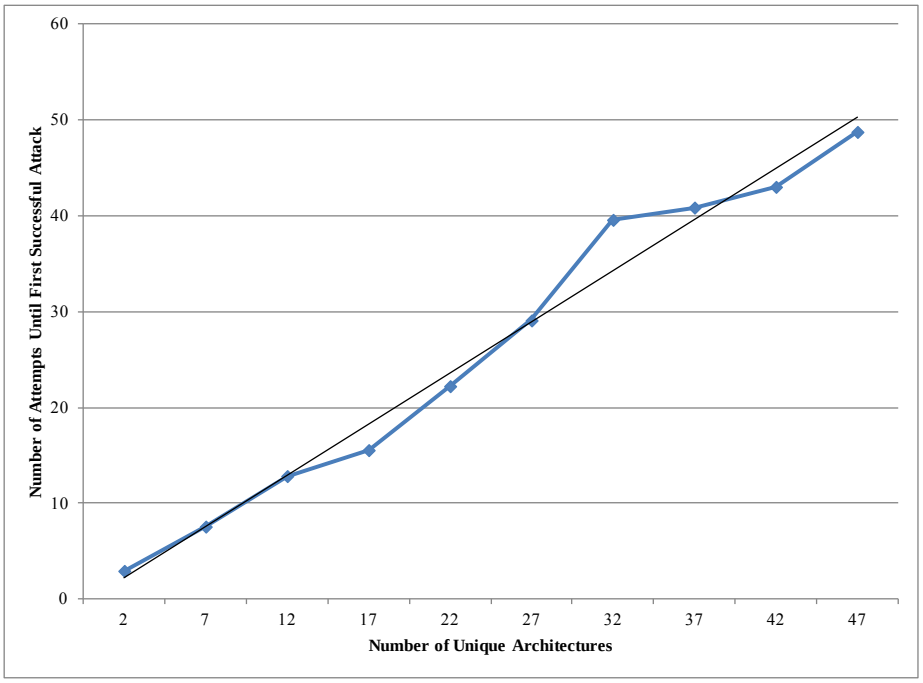

Figure 23. Instruction Encoding Average Attempts to Breach

The average number of attempts to successful breach is equal to the number of architectures. As such, the likelihood of successful attack is the inversely proportional to the number of architectures. Mathematically the inverse relationship corresponds to a geometric distribution as each one of the attack attempts can be viewed as Bernoulli trial.

Given that x is the attempt the attack will be successful and p is the probability of success (fixed for a given set of architectures):

$$x = geometric(p) \tag{1}$$

As x is geometric, the probability that the k^{th} trial is successful is given as:

$$P(x = k) = (1 - p)^{k-1} \times p \tag{2}$$

In addition, the expected value of a geometric distribution is given as:

$$E(x) = \frac{1}{p} \tag{3}$$

However p is known, so the equation reduces to:

$$E(x) = \frac{1}{\#\ of\ architectures^{-1}} = \#\ of\ architectures \tag{4}$$

Therefore, the experimental results from Figure 23 agree with the statistical equations (2) and (4). However, these numbers assume that no action is taken by the system operator when these attacks fail. As discussed in section A.2.1, when the attacks fail, there is a noticeable invalid instruction displayed to the operator. Given the severity of instruction-level errors it is likely that an operator (or similar security monitoring software/hardware) would recognize the incoming attacks and respond. By incorporating the "watchdog" the odds of the system being able to stop the attack are increased. Assume that after N unsuccessful attacks the system is locked down and access is no longer possible:

$$P(Attack\ prevented) = 1 - \sum_{0}^{N} P(x = i) \tag{5}$$

Equation (5) is using the geometric probably for each trial from (2). The watchdog approach was implemented in the experimental system for a number of 2 up to 47 architectures, with 25 runs experimental runs per data point.

Figure 24 shows the experimental results from the watchdog runs. Once the number of architectures reaches 30, the odds for successfully stopping an attack reach 80% and additional architectures only marginally increase that percentage. After this point, the added cost (in space and processing time) is likely not worth the return.

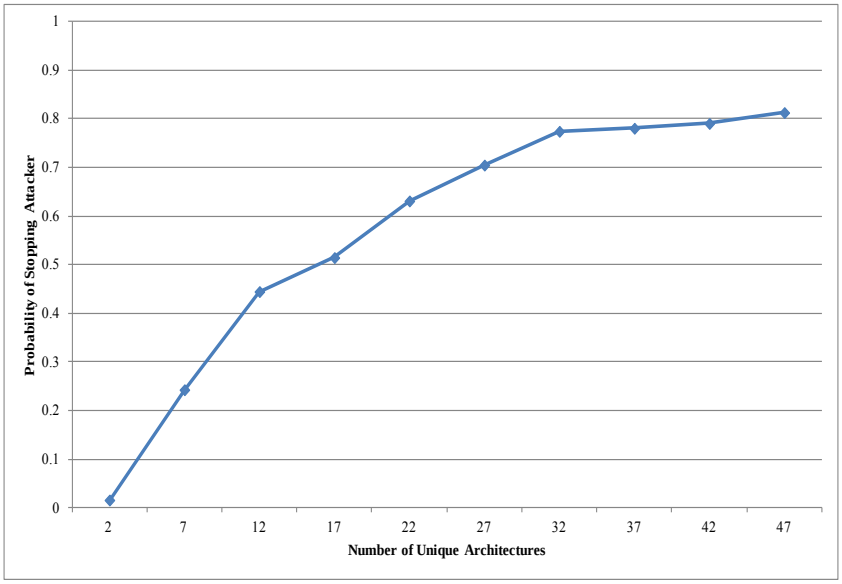

Figure 24. Probability of Preventing Attack with Watchdog

The statistical analysis assumes a worst-case scenario; the attacker is aware of the set of architectures that could possibly be run, and that attack payload that will be used only needs to run in a single architecture. If, for example, the architecture that was currently being executed switched during attack payload execution, the attacker would need to be aware of the change and construct the payload accordingly, making payload construction increasingly difficult.

Synthetic architectures provide another means for increasing diversity in execution. While synthetic architectures can provide reliable protection especially if a watchdog is available, they provide diminish returns as the number of available architecture increases. Similar to segmentation, the eventual selection of a single architecture ultimately limits the protection. To combat the limitations of single-architecture selection, another approach called architectural shifting was developed.

3.3 Reducing the Attack Surface via Architectural Shifting

A key finding from the investigation of application segmentation and synthetic architectures is that while increasing the number of architectures in a system does improve security, the effectiveness is limited because the attacker must only correctly guess a single architecture. Given the difficulty of including large number of architectures even if they are synthetic, executing on a single architecture ultimately limits protection even if there are many choices at runtime.

The key to preventing attack is incorporating many architectures during execution. Segmentation provides one approach for incorporating multiple architectures into application execution but is difficult to automate and is limited in granularity of the segmentation. Instead,

an approach called *architectural shifting* was developed that uses many architectures to execute an application. Architectural shifting is the computer architecture analogue to frequency hopping in radio systems where every instruction can be executed on one of many architectures. Many, shifting architectures ultimately force an attacker to craft a multi-architecture payload as well as appropriately guessing the proper sequence of architectures being employed. Not only is an adversaries attack effort increased but also the probability of successfully crafting a payload that operates in a shifting environment is near zero for even a small number of architectures and a small payload size.

From section 3.2.3 we know that the probability of an attacker successfully guessing the correct architecture (with a single architecture payload) is simply the inverse of the number of architectures. When multiple architectures are required for the payload, the probability of successful breach can be expressed as:

$$p = \frac{1}{a^r} \tag{6}$$

Where a is the number of possible architectures, and r is the number of architectures required in the payload. The statistical analysis again assuming that the attack knows the set of possible architectures (the worst case scenario). Figure 25 shows the exponentially decreasing probability of attacker success as the number of architectures required in the payload increase. In the example shown, only 5 possible architectures are available (architectures can repeat, which allows the number required to be higher than the number of possible).

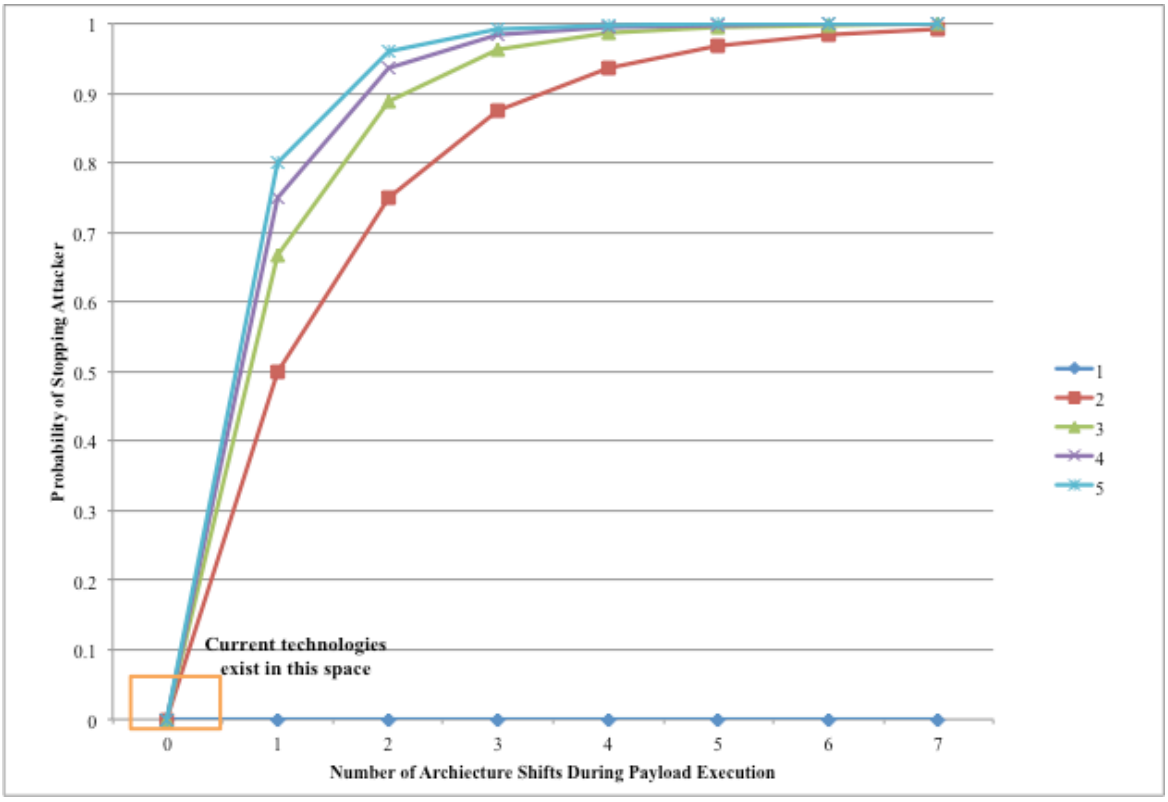

Figure 25. Probability of Stopping Attacker with Increasing Number of Architecture Shifts

Comparing the results of architectural shifting to the results from section 3.2.3, it is clear the security benefit that shrinking the attack space provides. Even with the smallest number of possible architectures (2), after only 5 architecture shifts, the odds of stopping the attacker are over 95%.

By increasing the number of architectures required in the payload, the faster the probability of stopping an attacker grows.

3.3.1 Implementation of Architectural Shifting

Architectural shifting leverages the advantages of instruction encoding to create a framework that is able to rapidly and randomly switch between different encoded architectures. Each set of uniquely encoded application instruction streams has the same base architecture, such that once they are decoded, each application instruction stream is identical. Therefore the processor can fetch instructions from any of the encoded applications, and provided it has the correct decode value, can decode and execute them. Instruction multiplexing can also be thought of as application segmentation at an instruction level.

Figure 26 shows the block diagram for instruction multiplexing. Stored with each set of instructions is its corresponding decoding value. The decoder/multiplexor selects one instruction stream, loads the decode value, and proceeds to fetch and decode. The multiplexor can switch

from stream to stream at any time interval (the smallest possible being switching after every instruction.) The tradeoff here is speed vs. security, since there is a setup time associated with each multiplexing transition. The example is shown with three sets of instructions, but functionally equivalent for any number of synthetic instruction sets.

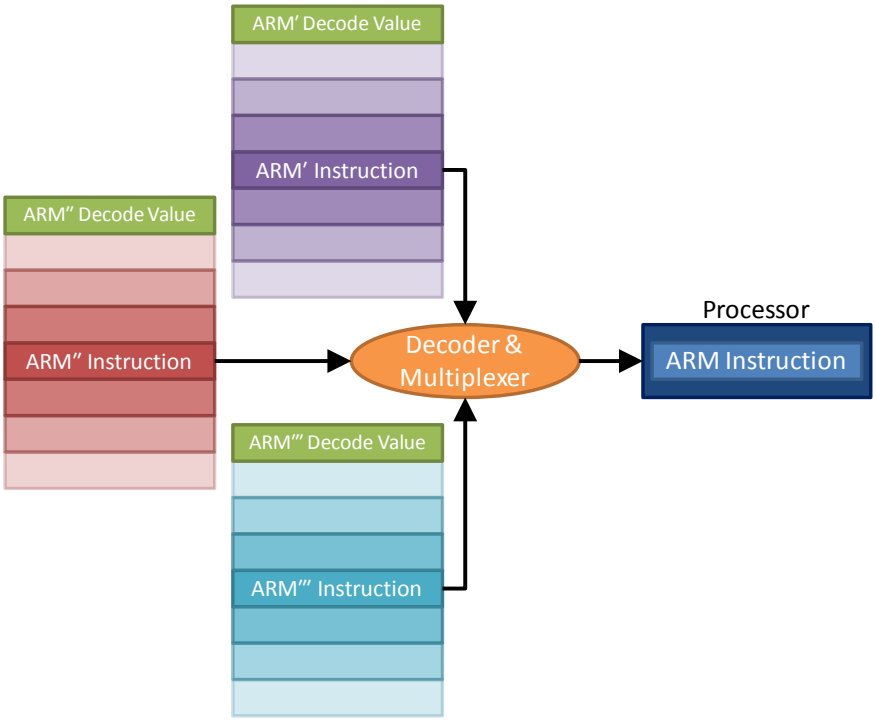

Figure 26. Instruction Multiplexing Block Diagram

The odds of attacker success against instruction multiplexing can be expressed as a function of the size of the attack payload in instructions (y), the number of instructions executed between multiplexer shifts (m) and the number of possible architectures (a):

$$p = \frac{1}{a^{\frac{y}{m}}} \tag{7}$$

Comparing (7) to (6), it is clear that the number of architectures required in the attack payload is equal to attack size divided by the multiplexer switching frequency ($r = \frac{y}{m}$). It should be noted that m is an approximate frequency since one instruction can vary in time required to execute. The attacker, whose goal is to keep r small, will try to keep y small. From the defenders perspective, it is best to have r as large as possible, and the defender will attempt to keep m small (1 is the smallest possible).

These variables allow a custom level of security to be set for specific situations. A system operator has control over both the number of possible architectures and the multiplexer shift frequency. Increasing the number of possible architectures, increases security, at a cost of

memory. Increasing the multiplexer shift frequency will also increase security, but at the cost of processing cycles. Figure 27 shows the increased probability of stopping the attacker with both an increased frequency of multiplexer shift and number of possible architectures.

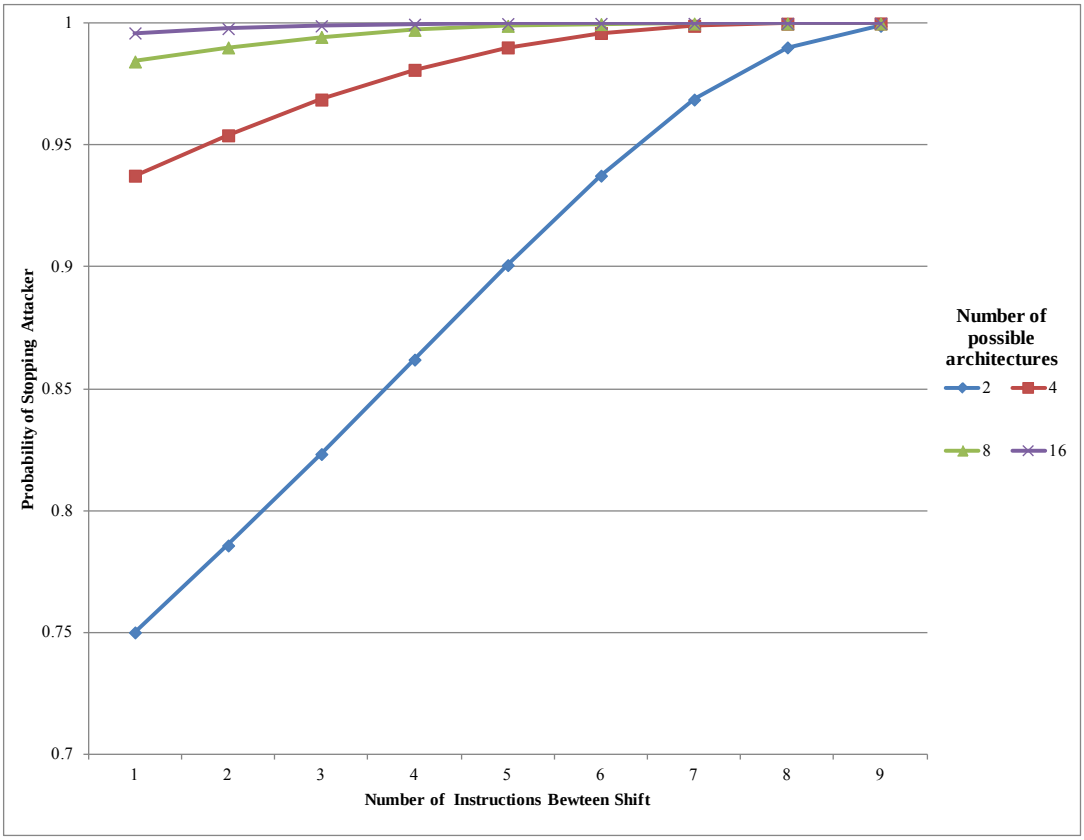

Figure 27. Probability of Stopping Attacker with Increasing Frequency of Multiplexer Shift

3.3.2 Instruction Multiplexing with QEMU

Similar to instruction encoding, instruction multiplexing exhibits similar implementation challenges as discussed in section 3.2.2. Namely, the decoder/multiplexor needs to be placed in the instruction fetch path between the processor and memory. Instruction multiplexing was implemented in the QEMU emulators, expanding on what was already done for instruction encoding.

When an application is started with instruction multiplexing in QEMU, multiple instruction sections are allocated in memory. The application is then encoded multiple times (using the same process detailed in section 3.2.2) and stored in the newly allocated memory. The instruction fetch process now chooses a random instruction stream as well as decoding the instruction. Since all the locations of the instruction streams are known by QEMU, when it is time to choose a new architecture, it is a simple matter of updating the MMU to point the virtual address to a new set of instructions.

3.3.3 Combined Results of Synthetic Architectures and Architectural Shifting

The final CHECKMATE protection capability consists primarily of the combination of many synthetic architectures and rapid, realtime architectural shifting. To validate the statistical analysis above, the attack scenarios and OpenSSH application described in previous sections were used to demonstrate the effectiveness of the finalized CHECKMATE solution. An experiment was setup where an eight instruction payload was used to attack a vulnerable OpenSSH application on a CHECKMATE protected platform with only two architectures. The aforementioned configuration represents a worst case scenario where the minimal protection is used and the smallest possible payload is required and thus represents a strong test case for evaluating CHECKMATE's effectiveness.

The screenshot in Figure 28 shows the experiment running in realtime. Note that the number of successful instructions (not complete attack) is very small even over millions of attempts even for the worst case configuration of a CHECKMATE protected system and attack. Experiment results support the previously discussed statistical model derived from the architectural changes imposed by CHECKMATE.

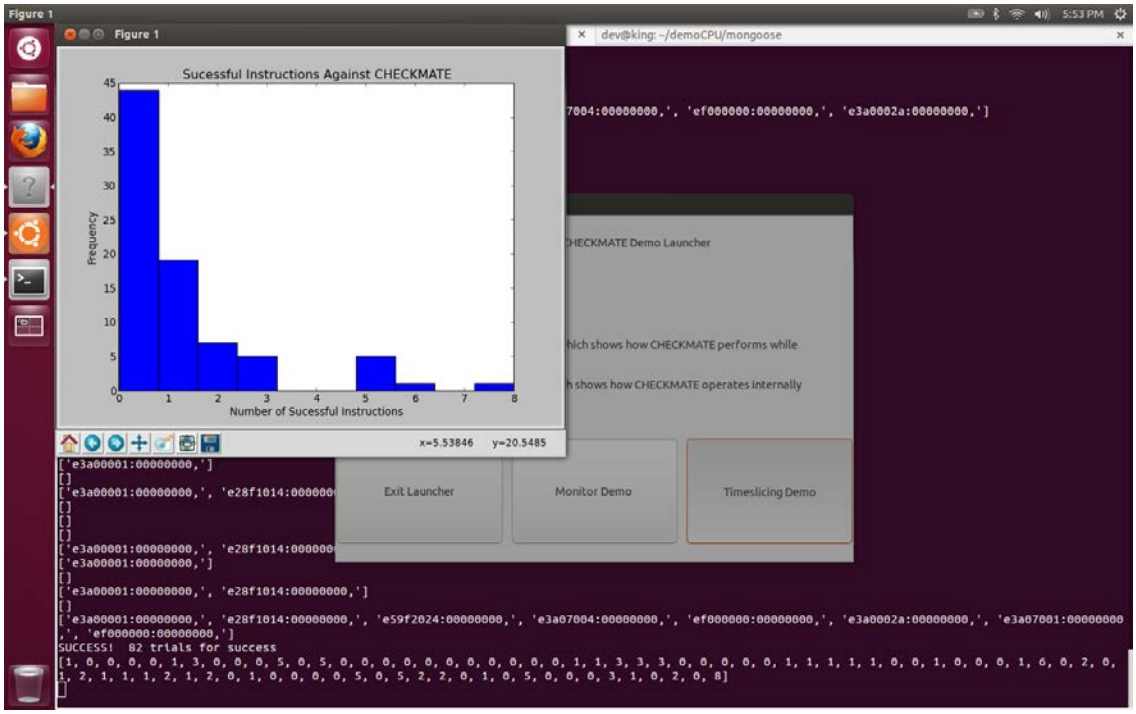

Figure 28: Realtime CHECKMATE Experimental Results

4 RESULTS AND DISCUSSION

4.1 Key Finding 1: Exponential Increase in Protection due to Diversity

Combining multiple architectures with realtime architectural shifting yields a drastic increase in protection by creating extreme diversity in application execution. The ability to achieve effective protection through diversity finding represents the core of the CHECKMATE protection capability. In some cases, application segmentation can be used to execute portions of application across heterogeneous, remote hardware assets to further diversify portions of applications. The CHECKMATE approach successfully leverages extra hardware to provide its protection including not only multiple, heterogeneous processors but also underutilized processors and unused memory. CHECKMATE is implementable as both a software or hardware solution and is applicable to any application with available source code. The CHECKMATE hardware solution provides additional robustness from attack, reduced performance impact, and the potential for applying CHECKMATE protection to entire operating system environment as well as standalone, bare-metal embedded systems.

4.2 Key Finding 2: Attack Detection and Alerting

An additional benefit of the CHECKMATE approach beyond protection from attack is that is illuminates the attack attempts by forcing attack attempts to manifest themselves as illegal/invalid instructions. Independent of the diversification technique employed; an attack being attempted in the incorrect architecture will produce an illegal instruction error. These methods can be instruction encoding, remote execution or time multiplexing. Attack attempts are easily detected by an operator or other automated means enabling realtime response and countermeasures deployment. This approach to attack detection is especially powerful because it occurs in response to an immutable property of the physical processors and cannot be disabled.

4.3 Key Finding 3: Attack Types Prevented

Table 2 provides a brief summary of CHECKMATE protection against major attack classes. In summary, CHECKMATE completely neutralizes an entire *class* of attacks, machine code injection attacks, that are commonly employed in attacking a wide range of computing systems. Attackers leverage vulnerabilities and flaws in applications combined with the knowledge of the underlying system hardware to craft and execute attacks. CHECKMATE neutralizes machine code injection attacks by removing a fundamental mechanism available to the attacker.

Table 2. CHECKMATE Protection Summary

Attack Type	Protection
Machine Code Injection	CHECKMATE able to protect against
Return Oriented Programming	CHECKMATE offers limited protection
Interpreted Code Injection/SQL Injection	CHECKMATE cannot protect against

When segmentation and remote execution are employed, CHECKMATE can provide limited protection against return oriented programming attacks (ROP).

Segmentation provides limited protection, simply because the attacker's selection of ROP gadgets is reduced, and it is not possible to reach across application boundaries to get other gadgets. Architectural shifting has little impact against ROP attacks since the ROP gadgets will be fetching and executing instructions in the same manner as a regular application

CHECKMATE does not provide protection against interpreted code injection attacks. Interpreted environments or virtual machines act as a barrier between the application and the native architecture. Interpreted environments have benefits in certain situations, because the same executable can run on multiple architectures. Similarly, attacks against interpreted environments are widely applicable across multiple architectures. Figure 29 shows an example an attack against an interpreted environment with a Java executable. CHECKMATE can also not protect against SQL injection attacks for the same reason; SQL commands are the same regardless of the architecture they run on.

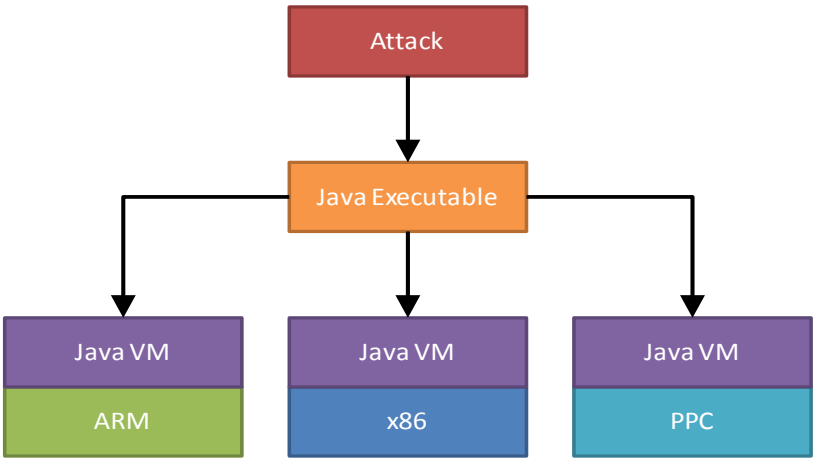

Figure 29. Interpreted Code Injection Attack on Multiple Architectures

4.3.1 User Experience and Performance

CHECKMATE is designed to work seamlessly within an existing system. For most instances, users operate their system without knowledge that CHECKMATE is even present. For example,

a user executes "/usr/bin/wget". Behind the scenes, the CHECKMATE system automatically applies protection by running in different architectures.

Given that most applications executed in a general purpose, user-interactive computing environment are not compute bound, a user generally experiences little or no discernible application slowdown. The primary source of delay in user interactive applications is system I/O. In the case of emulation, the cost of emulating many architectures far outweighs the cost of the CHECKMATE protections in both resources consumption and compute performance. Any performance lost can be recovered by the reduction in complexity of anti-malware software permitted by the reduction in the attack surface afforded by CHECKMATE.

One aspect a user may notice is increased memory usage. For the instruction multiplexing approach, CHECKMATE can be configured for a number of simultaneous synthetic architectures. The memory increase is equal to the product of the number of additional architectures and the amount of executable code in the program. Given the amount of memory found in commodity systems, the fact that much of memory is usually data and not instructions, and that only a few architectures are required to provide incredibly improved security, it is expected additional memory use is not a major limitation.

Another aspect a user may notice is the speed of executing the program. Because CHECKMATE emulates different processor architectures, some performance is lost during the translation to the host processor architecture. Due to the use of widespread emulation, some applications may experience up to 500% *compute* slow down depending on several factors. However, in most applications, slowdown is imperceptible to user and negligible to the application as actual computation represents a small portion of overall execution time. For users who desire the highest performance, CHECKMATE is also implementable in a hardware-based solution (see section 3.2.2.4).

5 CONCLUSIONS

CHEKMATE has been demonstrated to be an effective, low cost and reliable mechanism to greatly reduce the available cyber attack surface. These results are shown in laboratory experiments. If these results continue to be seen in a TRL 5 environment, CHECKMATE can contribute to a substantial reduction on the attack surface presented by modern DoD systems.

In an effort to combat widespread attacks against monoculture computing environments, Raytheon BBN Technologies (BBN) developed and demonstrated CHECKMATE (Hardware-assisted Platform Diversification for Secure Polymorphic Computing), as a novel set of protections to add diversity and break the monoculture. CHECKMATE neutralizes and entire class of attack that is commonly used to attack real world systems. The use of many

architectures combined with architectural switching represents a fundamental advancement in computer architecture design for security.

BBN has shown that in addition to stopping code injections attacks, increased diversity has the benefit of notifying the system operator when an attack occurs, due to the nature of an instruction architecture mismatch. Attack detection allows for the deployment of countermeasures or corrective actions.

6 RECOMMENDATIONS

6.1 Transition CHECKMATE to TRL5 and Deploy CHECKMATE at Small Scale in an Operational Environment

Having demonstrated the benefits of techniques in a laboratory environment, evaluation of CHECKMATE in a representative operational environment would provide a larger scale, experimental basis for quantifying CHECKMATE's impact on system protection. Additionally, exposing CHECKMATE to actual attacks or potentially a red team scenario provides additional means for evaluating the robustness of the CHECKMATE approach.

6.1.1 Example CHECKMATE Enterprise Deployment Scenario

CHECKMATE provides a new set of techniques for protection against threats in a monoculture-computing environment that have been demonstrated in a proof-of-concept environment. While adequate, several areas of improvement can be made to take CHECKMATE beyond the proof-of-concept.

The following sections details both the administrative and technical steps required to roll out CHECKMATE in an enterprise environment. A sample roll out is provided as a baseline for discussion. We discuss exploit scenarios, and demonstrate CHECKMATE protections against them. We also discuss the user experience performance, as well as the ability for CHECKMATE to act as an administrative alerting system. Finally, we discuss additional research topics that can further improve protection capability.

6.1.2 Sample Roll Out

The following details an example roll out into an enterprise system with 5 servers and many clients as shown in Figure 30. CHECKMATE protects the primary service running on each server. On the clients, a standard set of applications (e.g. "/usr/bin") is protected with CHECKMATE.

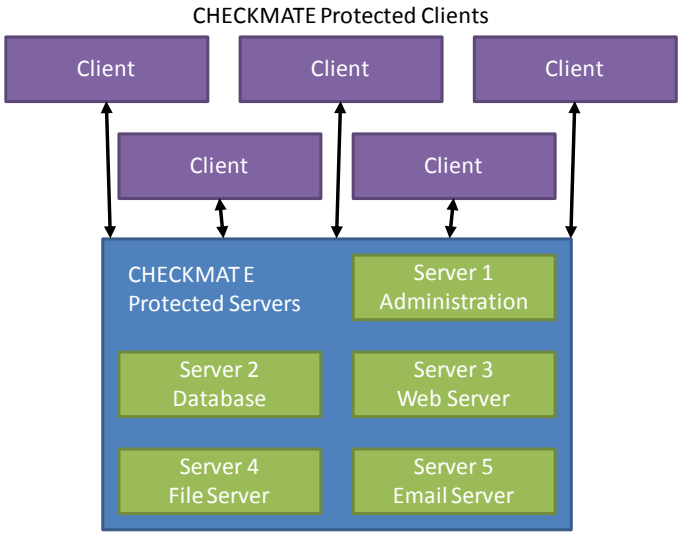

Figure 30. CHECKMATE Enterprise Deployment

Server 1 - Administration
Server 2 – Database (mySQL)
Server 3 – Web server (Mongoose)
Server 4 – File server (SSH)
Server 5 – Email server

Each service in the example network is exposed to the Internet. Since CHECKMATE protection works at the lowest level of instruction execution, it automatically provides both internal and external threat protection for these services. So even if a client on the inside of a network is compromised, CHECKMATE-enabled clients and server are still protected.

6.1.3 Active Exploit Scenarios

To further illustrate CHECKMATE protections, example attacks against the enterprise configuration described in the previous section are detailed. Table 3 depicts typical attack scenarios in an enterprise environment. CHECKMATE's ability to defend against typical enterprise attack scenarios is shown in Figure 31.

Table 3. Active Exploit Scenarios

Attack Scenario	Description
1. Malicious external client attacking public facing services	Perhaps the most probable scenario where an attacker external to the network is attacking a publicly facing service. For our example, this attack is likely against the web server.
2. Malicious external web server attacking a client	The clients inside the network have access to the Internet. If one of those clients visit an external malicious server, it is possible that they could become compromised.
3. Malicious internal client attacking internal services	It is also conceivable that an insider threat exists within the sample network. In this scenario, the insider threat seeks to attack the internal network services such as database and file servers.

The first scenario is perhaps the most likely of all of the three attack scenarios. An external malicious threat attempts to attack the external facing web server in the sample network. Hypothetically, the attacker is aware of a specific vulnerability within the web server, and has crafted an exploit against it. In our example, the web server is protected with CHECKMATE, and is able to defend against code injection attacks. When the attacker launches their attack, it is thwarted by CHECKMATE, and potentially, administrative notifications are generated (see section 6.1.4).

Similar to the first scenario the second scenario also includes and external attacker, however the internal client is threatened. The internal client is running a web browser vulnerable to several different code injection attacks. When the user of the internal client visits a malicious website, an attack is launched against their vulnerable web browser. With the CHECKMATE protection, again the attack is thwarted, and notifications are generated.

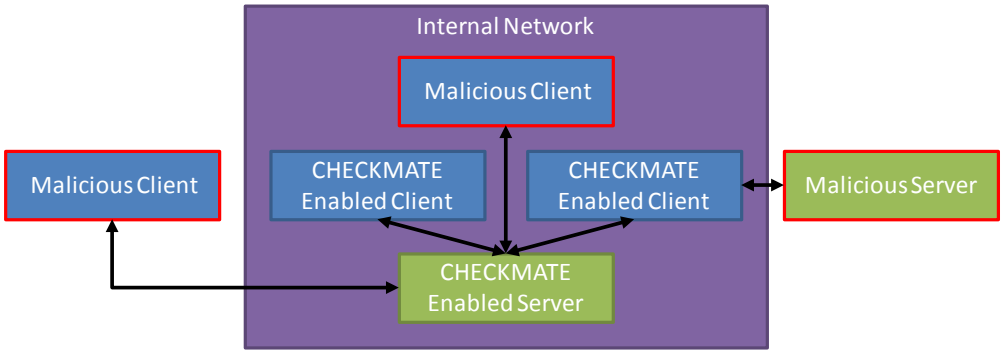

Figure 31. Active Exploit Scenarios

In the final scenario example, we revisit the first scenario, but now we have an internal threat against the services running within our sample network. The threat is generated from an internal source. Note that even though that attack is from a CHECKMATE-enabled client, CHECKMATE is not providing any type of protection at the *source of the attack*. However, the attack and protection provided is identical to the first attack scenario. Because the web server is CHECKMATE enabled, the attack is thwarted and administrative notifications are generated.

6.1.4 Alerting and Protection System

In addition to a proactive defense, CHECKMATE has the ability to provide an early warning system. During an active attack, an adversary attempts to exploit a specific vulnerability and inject malicious code. CHECKMATE provides a line of defense and is likely thwart injection attacks. Specifically, it is expected that the CHECKMATE system throws an exception rather than executing malicious code. An attack detection capability provides an opportunity to report the event to a system administrator, as well as gather statistics for later analysis. A previous section referred a monitoring technique as a "watchdog".

Deployment of a watchdog feature that would need to be further researched and developed. For the QEMU-emulated CHECKMATE environment, one possibility is to modify the emulation system to catch these processor exceptions and send to a notification system. The notification system could be programmed to allow for some number of exceptions at a certain rate. Once a threshold is exceeded, an alert could be sent to an administrator via an email or SNMP trap.

6.1.5 Autonomous Protection

Extending the capability to alerting an administrator, it is envisioned that CHECKMATE could be further enhanced to automatically, temporarily, shutdown parts of a system during an attack automatically, or with administrator assistance. For example, consider the web server in our sample network. If an attack threshold exceeds say, 10 per minute, CHECKMATE could shut down the web server for 10 minutes and sends an administrative alert. If after 10 minutes, the service is restored and attacks continue, CHECKMATE could permanently bring down the service.

6.2 Extend CHECKMATE to Operate on Additional Platforms

The CHECKMATE research and implementation is largely based on Linux systems due to the openness and flexibility afforded during the research process. Fundamentally, the CHECKMATE concept is applicable to a broad range of systems with proper extension. For example CHECKMATE could be extended to protect embedded systems such as routers or radios or other commodity general-purpose computers systems and applications such as systems using Microsoft Windows, and Mac OS. Any system that has the ability to sufficient resources

to emulate additional process architectures (e.g. QEMU), and executables can be recompiled cross-architecture, are candidates for the CHECKMATE technology.

6.3 Develop Advanced Techniques for Achieving Synthetic, At-Scale Diversity in Future Systems

Given the effectiveness of diversity for preventing an entire class of attacks, techniques for designing or incorporating diversity into next-generation systems should be explored. For example, randomization of platform design at both the software and hardware levels may provide additional protection. Also, developing additional techniques that provide systematic uniqueness at the hardware level both in construction and operation could provide a means for robust protection of future systems through diversity.

6.4 Additional CHECKMATE Implementation Extensions and Enhancements

Table 4 describes additional areas of research that could be performed to extend or enhance CHECKMATE performance.

Table 4. Additional CHECKMATE Research

Research Area	Description
ELF Loader	To perform load-time encoding (eliminating any pre-encoding requirements)
Automated Segmentation techniques	Methods to automatically, or semi-automatically segment a program along smaller boundaries. Research to date has included process boundaries, but preliminary research suggests this could also be done along functional boundaries.
Investigate GPU offloading	GPUs are essentially commodity processors with unique instruction sets. Research into utilizing this additional resource as well as extending the emulation diversity could result in further diversification enhancements.
Dynamic Link Library improvements	Research to improve handling of dynamic link libraries, since the runtime memory is shared amongst many processes, need to decode consistently.
Add full THUMB instruction support	And full ARM instruction set
Enhanced Demonstrations	CHECKMATE-Enabled web-browser CHECKMATE-Enabled PDF viewer

Enhanced Demonstrations	Perform exploit against known vulnerable code, with and without CHECKMATE protection enabled, and examine performance (security + speed)
CHECKMATE-aware MMU and enhanced decoder	Fast, software-less decoding which utilized the processors hardware MMU to switch between multiple versions of the instruction stream at regular intervals
Additional OS Support	To support Windows specific features and APIs/libraries

7 REFERENCES

[1] “OpenSSH,” [Online]. Available: http://www.openssh.com/.

[2] S. Lyubka, “Mongoose - easy to use web server,” [Online]. Available: http://code.google.com/p/mongoose/.

[3] J.-l. Gailly and M. Adler, “gzip,” [Online]. Available: http://www.gzip.org/.

[4] J. Smith and J. Ioannidis, Implementing remote fork() with checkpoint/restart, Columbia University.

[5] G. Portokalidis, “MINESTRONE: Identifying and containing software vulnerabilities,” [Online]. Available: http://nsl.cs.columbia.edu/projects/minestrone/?p=0.

APPENDIX

A.1 Initial Experimentation

For the initial goal of the CHECKMATE program; researching the security benefits and feasibility of using existing but underutilized hardware in a system, the target architectures of ARM, PowerPC (PPC), and Intel x86 were chosen. These processor architectures are fairly common in modern embedded systems, and have well established compiler support within a Linux environment.

The first technique evaluated consisted of choosing one of the three architectures when an application needed to run. It is assumed that the application that was going to be run had already been compiled for each of these architectures, and the executable already resided on those systems.

The hypothetical system consists of the components shown in Figure 32. The x86 systems is the interface to the user, and the remote ARM and PPC systems are connected to the x86 system via a network connection. The constructed setup was chosen both for its simplicity, and its similarity to existing systems that would benefit from CHECKMATE protection.

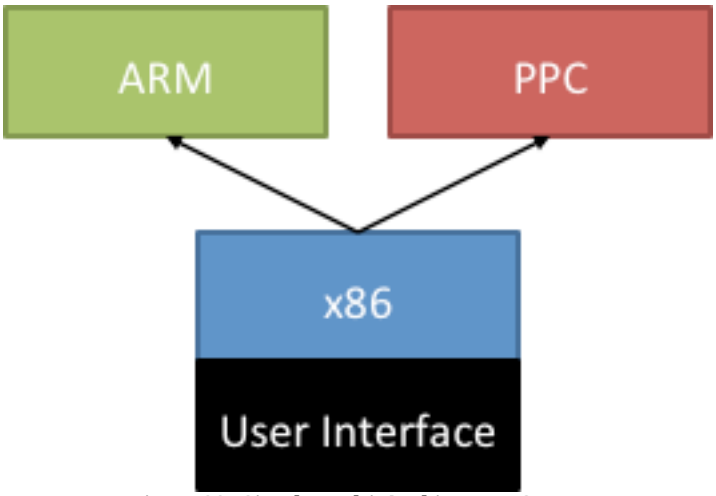

Figure 32: Simple Multi-Architecture System

While simple, the physical design posed new challenges. First, these processing systems need a way to communicate with each other. If a user started an application on one of the remote architectures, that application would require some of the local resources of the x86 architecture system (files, network interfaces, I/O). Secondly, the solution cannot be tied to a specific set of architectures. It is important to make the CHECKMATE solution scalable to many types of systems, since no two systems will be identical. Emulators allow multiple types of guest systems

be run on different host systems, so different emulator options were evaluated to overcome the challenge of system availability.

A.1 Synthetic Attack Development

The attack used to test the segmentation technique was created specifically for the synthetic vulnerability that was added to OpenSSH. By adding our own exploit to the test program, we were able to ensure that the worst case scenario was under test. This attack allows an arbitrarily sized attack payload to be sent over in the password field of the authentication data when a session is created. An improperly sized buffer on the server side of the application allows the payload in the password field to be executed, resulting in the user being authenticated with root privileges. .

Figure 33 shows SSH authentication under normal conditions. Figure 34 shows SSH authentication under attack from the test payload. The test payload rides in the password fields, and when executed replaces the username with root.

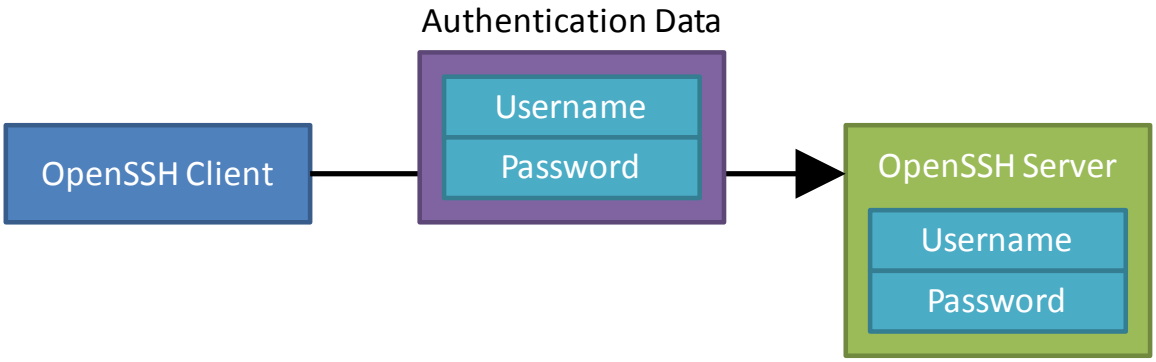

Figure 33. Normal SSH Authentication

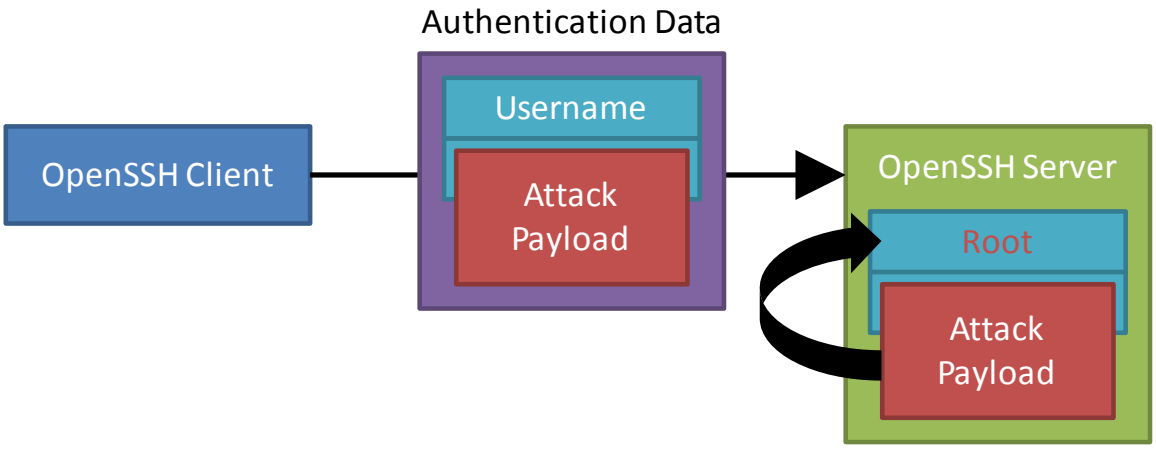

Figure 34. SSH Authentication Under Attack

A.2 Initial Findings

When attacking the test exploit in the initial test bed, several key findings became apparent:

A.2.1 Attack Indicator

When the architecture of the attack does not match the architecture of the running application, a very visible error occurs. This manifests itself as an illegal instruction, which can be observed by the system admin as an indication that an attack is occurring. The attack indication is a powerful, but previously unrecognized benefit of heterogeneous systems. While the primary goal of the types of attacks prevented by CHECKMATE is to gain access to system, a secondary goal is to circumvent detection mechanism. The architecture mismatch notification that CHECKMATE provides makes it difficult to do this. While illegal instructions will occur during the normal life of the system, it is easy to recognize multiple errors of this type, and take action.

A.2.2 Architecture Limitations

This approach with a limited number of architectures is not very effective. Using this setup, the attacker simply needs to retry the attack until the architecture of the attack matches the application. In this example, the attacker has a 1 in 3 chance of the attack being successful. When using this test bed, the odds of an attack being successful are proportional to the number of architectures in the system. To increase the odds of CHECKMATE successfully stopping an attack, more architectures need to be added to the system. Adding physical architectures to an existing system is rarely feasible, and there are a limited number of architectures that can be emulated. Making matters worse, there is a linear relationship to architectures added and the overall protection. Assume you include all 8 of the major architectures that QEMU is capable of emulating; the attacker still has a 1 in 8 chance of the attack being successful.

A.2.3 Application Limitations

Differences in applications also influence the effectiveness. Some applications, such as OpenSSH, are started on a server and run for a very long time. For applications like this, starting a random architecture is not effective, due to how long that instance might run. The design of an application influences how effective this approach is at improving security. For example, applications that have a long run time, do not see benefit from starting on a random architecture because an attacker will have ample time to try attacks using multiple architectures. In addition, system exploitation is not an exact science, and attackers are accustomed to having to try attacks multiple times to gain access to a system, so multiple attempts on a long running system is expected.

LIST OF SYMBOLS, ABBREVIATIONS AND ACRONYMS

AMBA – Advanced Microcontroller Bus Architecture

AXI – Advanced Extensible Interface

ELF – Executable and Linkable Format

FPGA – Field Programmable Gate Array

GNU – GNU's Not Unix!

GPU – Graphics Processing Unit

HTTP – Hypertext Transfer Protocol

IPC – Inter Process Communication

RISC – Reduced instruction set computing

ROP – Return Oriented Programming

SoC – System on Chip

SQL – Structured Query Language

SSH – Secure Shell

XOR – exclusive or